UNICORE Summit 2014

Proceedings, 24th June 2014 | Leipzig, Germany

Valentina Huber, Ralph Müller-Pfefferkorn, Mathilde Romberg (Editors)

JÜLICH
FORSCHUNGSZENTRUM

Mitglied der Helmholtz-Gemeinschaft

Schriften des Forschungszentrums Jülich

IAS Series Volume 26

Forschungszentrum Jülich GmbH
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

UNICORE Summit 2014

Proceedings, 24th June 2014 | Leipzig, Germany

Valentina Huber, Ralph Müller-Pfefferkorn, Mathilde Romberg
(Editors)

Schriften des Forschungszentrums Jülich

IAS Series Volume 26

ISSN 1868-8489 ISBN 978-3-95806-004-3

Bibliographic information published by the Deutsche Nationalbibliothek.
The Deutsche Nationalbibliothek lists this publication in the Deutsche
Nationalbibliografie; detailed bibliographic data are available in the
Internet at http://dnb.d-nb.de.

Publisher and Distributor:	Forschungszentrum Jülich GmbH Zentralbibliothek 52425 Jülich Phone +49 (0) 24 61 61-53 68 · Fax +49 (0) 24 61 61-61 03 e-mail: zb-publikation@fz-juelich.de Internet: http://www.fz-juelich.de/zb
Cover Design:	Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH
Printer:	Grafische Medien, Forschungszentrum Jülich GmbH
Copyright:	Forschungszentrum Jülich 2014

Schriften des Forschungszentrums Jülich
IAS Series Volume 26

ISSN 1868-8489
ISBN 978-3-95806-004-3

The complete volume is freely available on the Internet on the Jülicher Open Access Server (JuSER)
at www.fz-juelich.de/zb/openaccess

Persistent Identifier: urn:nbn:de:0001-2014111408
Resolving URL: http://www.persistent-identifier.de/?link=610

Preface

When the foundations of UNICORE were laid in the late 90's of the 20th century the "ancestors" intended to create a uniform interface to computing resources for a (small) number of computing centres. Today - in the era of eSciences and large distributed eInfrastructures - UNICORE has become one of the most innovative and major middlewares in Grid Computing serving users around the world.

The UNICORE Summit is a unique opportunity for users, developers, administrators, researchers, service providers, and managers to meet. Its objective is to exchange and share experiences, new ideas, and latest research results on all aspects of UNICORE. Since the first Summit in 2005, the organisers have received and reviewed a significant amount of distinguished contributions. Those selected and presented, complemented by invited talks, guarantee exciting Summits and lively discussions about the state-of-the art and the future of UNICORE, Grids, and distributed computing in general. The tenth edition, the UNICORE Summit 2014 has been held on 24 June 2014 in Leipzig, Germany.

The invited talk "HPC Applications in Biophysics, Material Science and Biomedicine - enabled by Unicore" by Borries Demeler, PhD, Associate Professor at the University of Texas Health Science Center at San Antonio, focused on applications of the UltraScan XSEDE Science gateway for high-resolution modelling of hydrodynamic experiments. He provided an overview of the integration of UNICORE into the gateway architecture in order to facilitate job submission and workflow management and discussed examples of science and discovery enabled by this implementation.

The next presentations highlighted an interesting use case using UNICORE workflow services for Polarize Light Imaging, new ideas and concepts for the future development of the UNICORE portal, experiences with certificate-free user-friendly HPC access based on LDAP with UNICORE and UNITY, perspectives for REST services in the UNICORE environment, integration of UNICORE services in a private cloud computing platform and resource scheduling algorithms in distributed problem-oriented environments.

The slides to the presentations can be found on the web at `http://www.unicore.eu/summit/2014/schedule.php`

We would like to thank all contributors for their presentations and papers as well as the organisers of the UNICORE Summit 2014 in Leipzig for their excellent work. Our deepest thanks go to the program committee members and reviwers for their hard work in reviewing papers. We are sure every participant will keep this wonderful event in mind. More information about the UNICORE summit series can be found at `http://www.unicore.eu/summit/`.

We are looking forward to the next UNICORE Summit 2015!

October 2014

Valentina Huber
Ralph Müller-Pfefferkorn
Mathilde Romberg

Program Committee

Valentina Huber (*Forschungszentrum Jülich*, Germany)

Piotr Bała (*ICM Warsaw University*, Polen)

Giuseppe Fiameni (*CINECA*, Italy)

Ivan Kondov (*Karlsruhe Institute of Technology*, Germany)

Daniel Mallmann (*Forschungszentrum Jülich*, Germany)

Ralph Müller-Pfefferkorn (*Dresden University of Technology*, Germany)

Mathilde Romberg (*Forschungszentrum Jülich*, Germany)

Bernd Schuller (*Forschungszentrum Jülich*, Germany)

Contents

Preface
V. Huber, R. Müller-Pfefferkorn, M. Romberg **i**

A Workflow for Polarized Light Imaging Using UNICORE Workflow Services
Björn Hagemeier, Oliver Bücker, André Giesler, Rajveer Saini, Bernd Schuller **1**

Fostering the Adoption of UNICORE Portal
Krzysztof Benedyczak, Piotr Bała, Marcelina Borcz, Valentina Huber, Rafał Kluszczyński, Mariya Petrova, Piotr Piernik, Bernd Schuller **15**

Certificate-free User-friendly HPC Access with UNICORE
Richard Grunzke, Ralph Müller-Pfefferkorn **23**

Perspectives for RESTful Services in the UNICORE Services Environment
Bernd Schuller, Jedrzej Rybicki, Krzysztof Benedyczak **31**

Providing Integration of UNICORE Services in Private PaaS Platform
Gleb Radchenko, Dmitry Savchenko **39**

Resource Scheduling Algorithm in Distributed Problem-Oriented Environments
Anastasia Shamakina, Leonid Sokolinsky **49**

A Workflow for Polarized Light Imaging Using UNICORE Workflow Services

Björn Hagemeier, Oliver Bücker, André Giesler, Rajveer Saini, and Bernd Schuller

Jülich Supercomputing Centre,
Research Centre Jülich, 52425 Jülich, Germany
E-mail: {b.hagemeier, o.buecker, a.giesler, r.saini, b.schuller}@fz-juelich.de

Understanding the anatomical structure of the human brain on the level of single nerve fibers is one of the most challenging tasks in neuroscience nowadays. In order to understand the connectivity of brain regions (affecting the brain function) on the one hand and to study neurodegenerative diseases on the other hand, a detailed three-dimensional map of nerve fibers has to be created. One technique applied to histological sections of postmortem brains is Polarized Light Imaging which allows the study of brain regions with a resolution at sub-millimeter scale. It is based on an optical property referred to as birefringence of myelin which surrounds the axons of nerve fibers. Therefore about 1500 slices, each 70 micron thick, of the post-mortem brain are imaged with a microscopic device using polarized light.

The images of brain slices are processed with a chain of tools for calibration, independent component analysis, enhanced analysis, stitching and segmentation. These tools have been integrated in a UNICORE workflow, exploiting many of the workflow system features, such as control structures and human interaction. Prior to the introduction of the UNICORE workflow system, the tools involved were run manually by their respective developers. Thus, once one step in the process was finished, the developer of the next tool in the chain would retrieve the data and run his tools on the output of the former. This manual approach led to delays in the entire process.

The introduction of the UNICORE workflow system for this particular use case resulted in several benefits. First of all, the results are easier to reproduce now, as fewer manual steps are involved. Secondly, the makespan of the entire workflow could be reduced to hours rather than weeks, because of the almost fully automated workflow. Lastly, only the automated approach will allow for the timely analysis of a large number of brain slices that are expected to be available in the near future.

This workflow is interesting from the technical point of view, as it takes UNICORE and its workflow system to the limits. Workarounds were required for some peculiarities of the workflow system. For example, in order to use results of one workflow job as input in the next job, the workflow system usually copies this data to the central workflow storage before copying it into the working directory of the next job. The amount of data for a single brain slice is on the order of magnitude of up to 1TB, with intermediate results at the same scale. Thus, the total amount of data easily adds up to several TB of data movement within the workflow, which can and should be avoided.

This paper will describe the situation as of version 6.6.0 of the workflow system. Results of this work have been incorporated in subsequent versions starting with 7.0.0. However, some of the approaches for processing large sets of data used here will still apply in future versions of the UNICORE system.

1 Introduction

Three Dimensional Polarized Light Imaging (3D-PLI)[3] has been developed at the Institute for Neuroscience and Medicine, Forschungszentrum Jülich. It is a technique applied to histological sections of postmortem brains, in order to create a detailed, three-dimensional

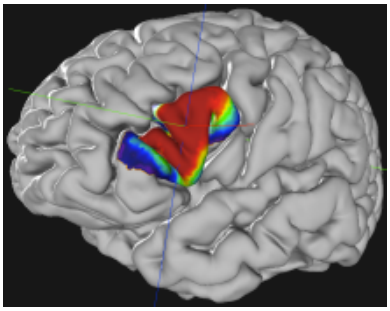

Figure 1. Fiber Orientation Map. The final result of the workflow.

map of nerve fibers of the brain. This will help understand the connectivity of brain regions (affecting the brain function) on the one hand and to study neurodegenerative diseases on the other hand.

The following is an overview of the tools involved in the workflow.

Image Calibration All images taken with the microscope need to be calibrated with average calibration frames to compensate for effects such as thermal effects of the CCD, camera electronic noise, and readout noise that all lead to signal variations obvious at each pixel[5].

Independent Component Analysis "The application of Independent Component Analysis (ICA) to PLI enables the effective separation of birefringence signals of myelin sheaths from noise and artifacts[5]. [...] In particular fibers with extreme inclinations (flat or steep) require high accuracy and sensitivity in signal sampling, in order to separate fibers with similar inclination angles, as well as to segregate regions of weak myelination from those with almost no myelination."[4]

PLI Analysis This is the actual analysis of fiber orientations within the brain slice. It is applied to overlapping tiles of the slice. The information from this step will later be reassembled in the stitching step.

Segmentation "Objects outside the tissue, e.g., residues from section preparation or dust particles are automatically segmented and removed from the image before the data sample enters the final pre-processing step."[4] Segmentation also separates brain tissue from the background at the border of the slice.

Stitching This step assembles information from the individually processed tiles in the previous steps into fiber orientation information of the entire slice.

Fiber Orientation Map The Fiber Orientation Map is the final result of the PLI technique. A visual example of such a map is displayed in Figure 1.

A full description of all the involved tools and codes is yet to be published. In this paper, we would like to put the emphasis on the complex workflow and the requirements it poses against the UNICORE workflow system.

The various tools involved in the process had originally been executed manually. There was a usual practice of transferring or copying intermediate data among working directories of tool developers, which poses a problem for the amount of data involved in the process. This approach was improved substantially by making versions of the tools available in well-known locations on machines at the Juelich Supercomputing Centre (JSC). Once deployed, these applications were embedded in a UNICORE workflow that takes care of invoking them in the right order on the respective machines, as well as handling input, intermediate, and output data during the runtime of the workflow.

2 Requirements of the Workflow

The process described above poses some requirements to the workflow engine, which are the subject of this section.

2.1 Parameterization

A workflow as a whole can be seen as an application on its own. Therefore, it is sensible to add parameters to easily take control over each execution of the workflow. This is particularly true for complex workflows, where parameters can help to tailor the workflow for a particular set of machines for execution.

The PLI workflow can be run with and without employing parts of the workflow. For example, the ICA step is optional. Also, data transfers among machines can be tuned to either use a shared file system or UNICORE file transfers in the absence of shared file systems. This can be controlled via an appropriate parameter. It is also a good idea to reference the actual data set to work on via a workflow variable, to avoid the need to modify individual jobs of the workflow for each workflow submission.

2.2 Iterating over Arbitrary File Sets

A brain slice in the workflow is comprised of tiles. The number of tiles belonging to a single brain slice or their names are not known before workflow execution. All tiles belonging to a slice are put in a directory, serving as input to the workflow. Independent jobs shall be created for each of the tiles as well as intermediate data. This is possible through the UNICORE workflow engine as described in section 3.4.

2.3 Data Movement

The UNICORE workflow system has a specific way of dealing with data and passing it from one workflow job to the next. It uses a central workflow storage, resulting in relevant results and data to be transferred or copied at least twice in order to pass them from one workflow job to the next. The first transfer will be from the job's working directory to the central workflow storage, the second transfer will be from the central storage to the next

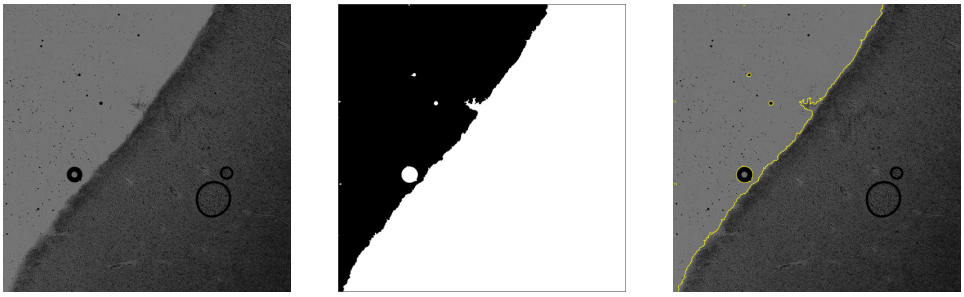

Figure 2. Seeded region growing is used to segment brain from background. On the left, we see the original image. The white area in the center image depicts brain matter. The border between brain and background is shown in the image on the right.

job in the workflow. There may even be situations where data is needed in more than just one subsequent job, increasing the number of transfers and copies of the data even further.

Raw Data of a single brain slice for the PLI workflow can consist of up to 1TB of image data to be processed by multiple jobs within the workflow. Intermediate results of the workflow jobs are of similar size and also need to be passed between jobs. Considering this amount of data, it is evident that data transfers should be reduced to the minimum. How this is achieved will be described in section 3.2.

2.4 Workflow Interaction

Part of the PLI workflow is dedicated to the segmentation of brain matter from the background. This is possible, because brain matter has a darker appearance in the image than the surrounding background. Segmentation involves a method called seeded region growing.[1,2] Seeds can be automatically put in the image based on pixels' grey values. However, the attribution of individual pixels to brain matter or background is based on a threshold provided by the user. The difficulty lies in determining a threshold between dark and light areas. This is done by considering a histogram of grey values of the entire image and setting the threshold accordingly. As the threshold is determined by the user, the workflow will be paused for the user to provide new values, potentially reiterating the selection process several times and carefully checking the intermediate result of the segmentation. Figure 2 shows the results of region growing on a sample image.

We will describe in section 3.5 how such interaction can be achieved with the UNICORE workflow engine.

2.5 Housekeeping Jobs

In complex workflows, a number of jobs for preparing input data, interpreting results, or taking decisions about further processing may be required. We call these housekeeping jobs. For some of these, it may be possible to run them as pre- or post-commands along with the actual job. In other cases, for example when the housekeeping jobs become more complex, this may not be sufficient, and extra jobs or scripts may be required.

In the PLI workflow, these jobs are used for several reasons. First of all, to initially setup a central directory to be used by the workflow jobs. Also, some jobs require preparatory work that is done in such housekeeping jobs. If multiple machines are in use, data may need to be copied among them, if no shared file system is available. This is also achieved in such a job.

A property that all these jobs have in common is that they do not actually contribute to the results of the computation, but are required to achieve the final result. They usually do not consume many resources by requesting multiple cores of the supercomputer and should complete as quickly as possible. Therefore, it is desirable to keep the time to completion of these jobs as short as possible. This will be the subject of section 3.3.

2.6 Workflow Portability

The PLI workflow must have the ability to be portable so it can be used on different supercomputers, file systems and infrastructures. Users have generally only a limited access to high performance supercomputers. After running the workflow applications on an HPC machine for a limited project duration, they are often forced to port the used workflow to another supercomputer or even back on their local clusters. Provided that the used image processing applications are already installed on the machine, the porting procedure should have a minimal impact to the workflow end-user. It should therefore be sufficient to edit only target systems, storages and the file transfer mechanism in the graphical user interface of the Unicore Rich Client.

3 Implementation

Some of the requirements mentioned in the previous section deserve special attention, as it is not quite obvious how to implement them in a UNICORE workflow. In particular, these are the iteration over arbitrary file sets, which is described in section 3.4. Additionally, the avoidance of data movement, which is described in section 3.2. Lastly, user interactions with the workflow are described in section 3.5. However, we will start by describing workflow parameterization, as they are the most obvious feature to use.

3.1 Parameterization

UNICORE workflows can be parameterized by introducing a number of workflow variables, the values of which control the execution of the workflow during runtime. An example of this, specific for the PLI workflow, is depicted in Figure 3.

In the PLI workflow, parameters control whether certain tools are used or not or how particular data transfers are done. There are four workflow variable declarations in the figure. Their meanings are:

RAW-DATA A String value pointing to the file system path of the raw data to be used.

DO-ICA A String value determining whether the Independent Component Analysis (ICA) shall be performed or not. Later in the workflow, the variable is evaluated in an IF control statement that only does the ICA if the value of this variable is “true”.

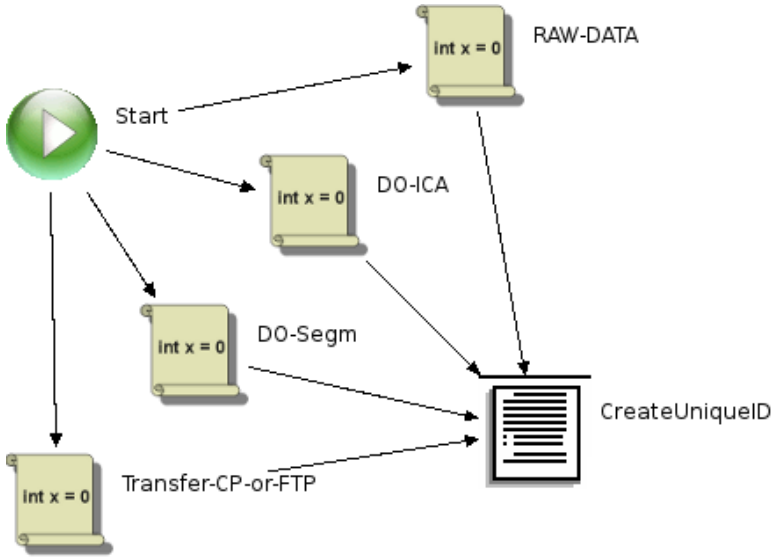

Figure 3. Workflow variables help to parameterize the workflow and control its execution during runtime.

DO-Segm A String value determining whether Segmentation shall be performed or not. Later in the workflow, the variable is evaluated in an IF control statement that only does Segmentation if the value of this variable is "true".

Transfer-CP-or-FTP This variable is used to control which transfer mechanism shall be used for the transfers among various machines. With the given set of machines available at JSC, some have common file systems and some do not. Whenever there is no common file system among machines, full UNICORE transfers are employed to transfer data from jobs on one machine to subsequent jobs on the other.

Additional variables can easily be conceived. For example, the common storage and its mount point to be used for storing intermediate files of the workflow can be introduced as parameters of the jobs. Section 3.2 on the implementation of data movement will provide more information about this.

3.2 Data Movement

In order to overcome the need for transferring all data from one job's working directory into the next job's working directory, we work with central workflow storage that is also available in the file system of the machine running the job. Therefore, we use a storage that is available in the machine's file system.

Additionally, if there are shared file systems among multiple machines at a single site, this may also be reflected by the configured storages. Thus, one will be able to exploit the file system for sharing data among multiple tasks within a workflow.

The setup of the central working directory, and the passing of information to subsequent jobs is achieved by means of housekeeping jobs, which are subject of the next section.

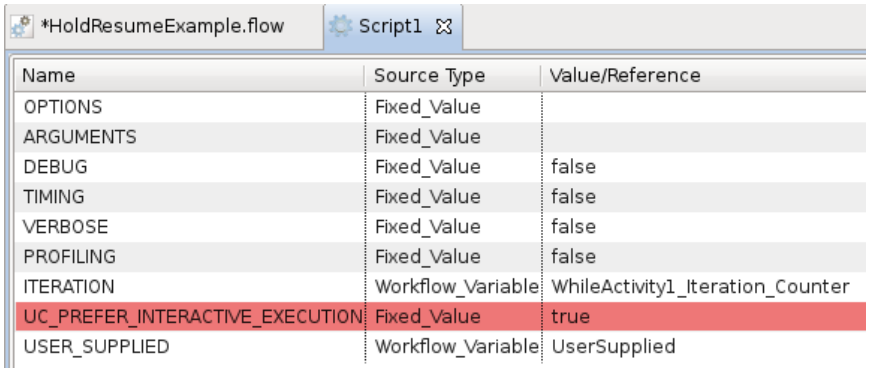

Figure 4. Setting UC_PREFER_INTERACTIVE_EXECUTION to true for a workflow job will cause it to be executed on the login node of a cluster rather than through the batch system.

There, we also provide the listings showing how to make use of the central workflow working directory. Section 3.4 also contains an example of how to avoid data movement while iterating over a set of files.

3.3 Housekeeping Jobs

Housekeeping jobs usually have a short runtime. It is therefore desirable and can be justified to run them outside the batch system, avoiding the queueing time.

This can be achieved by adding a variable called `UC_PREFER_INTERACTIVE_EXECUTION` with a value of `true` to the workflow job in question. This will be interpreted by the UNICORE server and TSI, and run the job on the head node of the machine. Figure 4 shows where to set this in the UNICORE Rich Client (URC)[8].

An example for such a housekeeping job is one that creates a unique directory for the execution of the workflow. This directory will be used by jobs to work on and exchange data, thus avoiding large data transfers among workflow jobs. The motivation for this has been given in section 2.3. In order to create this directory, we employ a script job containing the following lines.

```
1  tmp=$USER.XXXXXXXX
2  WFDIR=$(mktemp -u ${tmp}) || { echo "Failed␣..." }
3  echo WFDIR=${WFDIR}>PLI_unique_workflow_dir.txt
```

The name is written to a file named PLI_unique_workflow_dir.txt as a pair of key and value. The file is defined among the files to be exported from this job. In consequence, rather than transferring the actual data among many workflow jobs and potentially creating multiple copies of the same files, only references to the data are copied, substantially decreasing the required disk space and transfer time. Additional information can be added to the file as well. In the particular example of the PLI workflow, the initial job also extracts some pieces of information from the input data set that are later used as arguments to the invocation of one of the tools.

The following lines show the extraction of information specific to the input data set. These are also written to the same file for later reuse by other jobs.

```
5  ############## Getting the Data-specific information ##############
6  echo FILENAME=`ls ${DATA}/tiff/ | head -n1 | sed
       's-[0-9]*_\(.*\)_s[0-9]\{8\}.*-\1-' >>
       PLI_unique_workflow_dir.txt
7  echo SLICE_START=`ls -1 ${DATA}/tiff/ | grep -o "_s........_" |  cut
       -c 3-6 | head -n1` >> PLI_unique_workflow_dir.txt
8  echo SLICE_STOP=`ls -1 ${DATA}/tiff/ | grep -o "_s........_" | cut -c
       3-6 | tail -n1` >> PLI_unique_workflow_dir.txt
9  echo X_START=`ls -1 ${DATA}/tiff/ | grep -o "_s........_" |  cut -c
       7-8 | head -n1` >> PLI_unique_workflow_dir.txt
10 echo X_STOP=`ls -1 ${DATA}/tiff/ | grep -o "_s........_" | cut -c 7-8
       | tail -n1` >> PLI_unique_workflow_dir.txt
11 echo Y_START=`ls -1 ${DATA}/tiff/ | grep -o "_s........_" | cut -c
       9-10 | head -n1` >> PLI_unique_workflow_dir.txt
12 echo Y_STOP=`ls -1 ${DATA}/tiff/ | grep -o "_s........_" | cut -c9-10
       | tail -n1`  >> PLI_unique_workflow_dir.txt
13 echo GEN_DATE=`ls ${DATA}/tiff/ | head -n1 | sed
       's-\(.\{8\}\)_.*_s[0-9]\{8\}.*-\1-' >>
       PLI_unique_workflow_dir.txt
```

In order to understand the above lines, one has to know the structure of file names of the input files.

```
1  20131121_Brain11012_Temporal_L_70mu_40ms_s06920000_a00_d000_t130.tif
```

The beginning of the file name is a time stamp, followed by a brain identifier, region within the brain, and some technical parameters. The part starting with _s denotes the four digit slice number and the tile's position encoded as x and y coordinates with two digits each.

Consequently, the values of the above listed variables would become the following. Obviously, this is only a limited example as compared to multiple sections or even a full brain.

```
1  FILENAME=Brain11012_Temporal_L_70mu_40ms
2  SLICE_START=0692
3  SLICE_STOP=0692
4  X_START=00
5  X_STOP=04
6  Y_START=00
7  Y_STOP=04
8  GEN_DATE=20131121
```

Using these values in subsequent jobs is achieved by defining the workflow file exported from the first jobs as input and sourcing its contents as part of the scripts. This will make variables available for further usage.

```
1  source PLI_unique_workflow_dir.txt
```

3.4 Iterating over Arbitrary File Sets

The UNICORE workflow system supports the iteration over file sets, starting an interaction and jobs inside it for each file in the set. File sets can be defined either by listing the files directly when defining the workflow or by writing the list of files into a file that will be used. The latter approach allows for dynamically creating the list in a job preceding the loop and then use this list to run as many iterations as required.

The file list within the file must follow a certain pattern: to be recognized by the workflow engine, full storage URLs need to be listed.

```
1  ############## Determine workflow working directory name
       ##############
2  source CreateUniqueID_PLI_unique_workflow_dir.txt
3  USPACE=`pwd`
4
5  ############## Generate file list ##############
6  cd ${WORK}
7  FILELIST=`find PLI/${WFDIR}/caliOutputDir -type f`
8
9  for file in ${FILELIST}; do
10   echo 'BFT:https://fzj-unic.fz-juelich.de:9112/FZJ_JUDGE/services/'\
11 'StorageManagement?res='${USER}'-WORK#'${file} >>
       ${USPACE}/generated.txt
12 done
```

Whereas in this listing the storage URL is fixed, it could also be introduced to the job via a workflow variable, making the workflow more flexible and adaptable to new environments. However, with the current implementation of the workflow editor, this information needs to be provided in a hard coded fashion. The precise storage URL and mount point of the respective storage can be determined in the URC by checking the details of the Grid storage node in the Grid Browser as shown in Figure 5. Correlating these two items, Storage URL and mount point, is important to map the abstract UNICORE world into the concrete, incarnated environment of the job. Naturally, when intending to use a storage file system directly within a job, it must be available on the system where the job is executed.

Usually, jobs within workflow iterations as described in this section would define each of the files as input files. However, as described in section 3.2, this may not always be desirable. In the PLI workflow, the iteration mechanism is merely used to trigger the iterations, but not using the file import feature for the individual jobs of the iteration, thus avoiding duplication of files and transfers. Rather than importing the individual files, we import the list of files and extract the actual file name from it using the index of the current iteration.

```
1   ############## Determine which file to work on ##############
2  UnicoreStoragePath=`sed -n -e "$((FILE_INDEX+1))p" generated.txt`
3  PathFromWork=`echo ${UnicoreStoragePath} | sed -e 's/.*#//'`
4
5  ############# Define required files #############
6  Image=${WORK}/${PathFromWork}
7  Mask=$DATA/mask/`basename ${PathFromWork} .nii`_Mask.nii
```

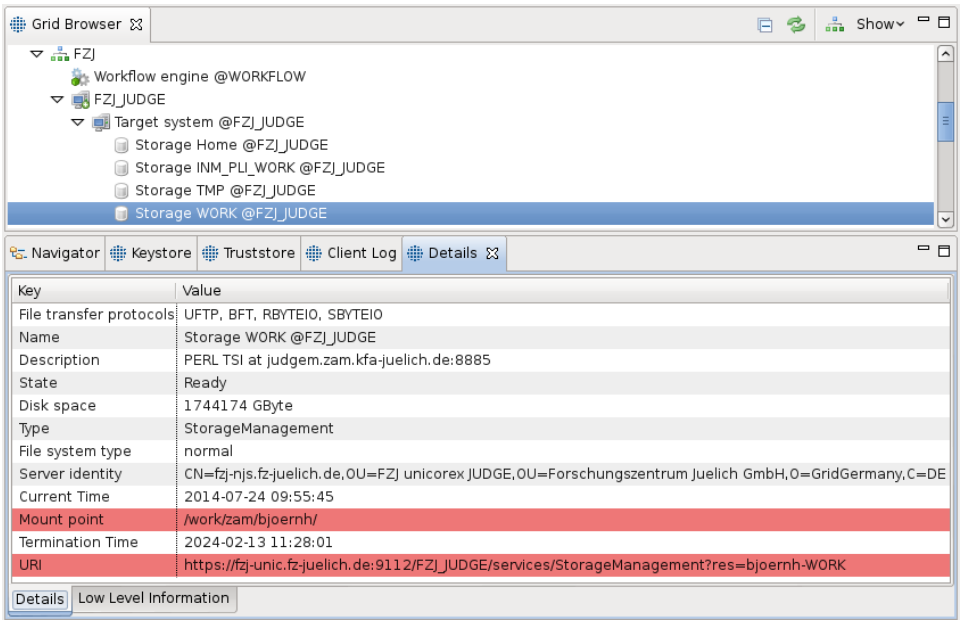

Figure 5. Storage URL and Mount point are available from the detailed information about a storage node selected in the Grid Browser. This information needs to be copied manually, it is not available in the workflow editor.

In this code snippet, the full storage path of the actual job's input file is extracted from the list file generated.txt. The FILE_INDEX variable is initialized for each job as the CURRENT_ITERATOR_INDEX workflow variable. The file generated.txt is the same file as the one used by the workflow engine to trigger the iterations. It is defined as an input file for the job.

The second line of the snippet extracts the path relative to the storage root by ignoring everything up until and including the hash mark. The full path to the target file is then constructed from the storage root, $WORK in this case, and the relative path. In order to be more flexible regarding the choice of storages and allow for defining the storage to use at the workflow level, a workflow variable could be imported into the job, too. In the listing above, WORK is a known environment variable on the machine that the job is intended to run on. It contains the path of a working directory. At the workflow level, the storage in use must then be the storage corresponding to this directory.

3.5 Workflow Interaction

As described above, there are situations where users require control over the further execution of the workflow. It has been possible to hold and resume UNICORE workflows in the past. The feature was introduced in the workflow engine with release 6.4.0 in March 2012. General support for using this feature has been available in the URC since its 6.6.0 release in July 2013.[8]

The interfaces for resuming workflows that are on hold in the workflow engine had already foreseen to pass parameters to the resume action and thus modifying workflow

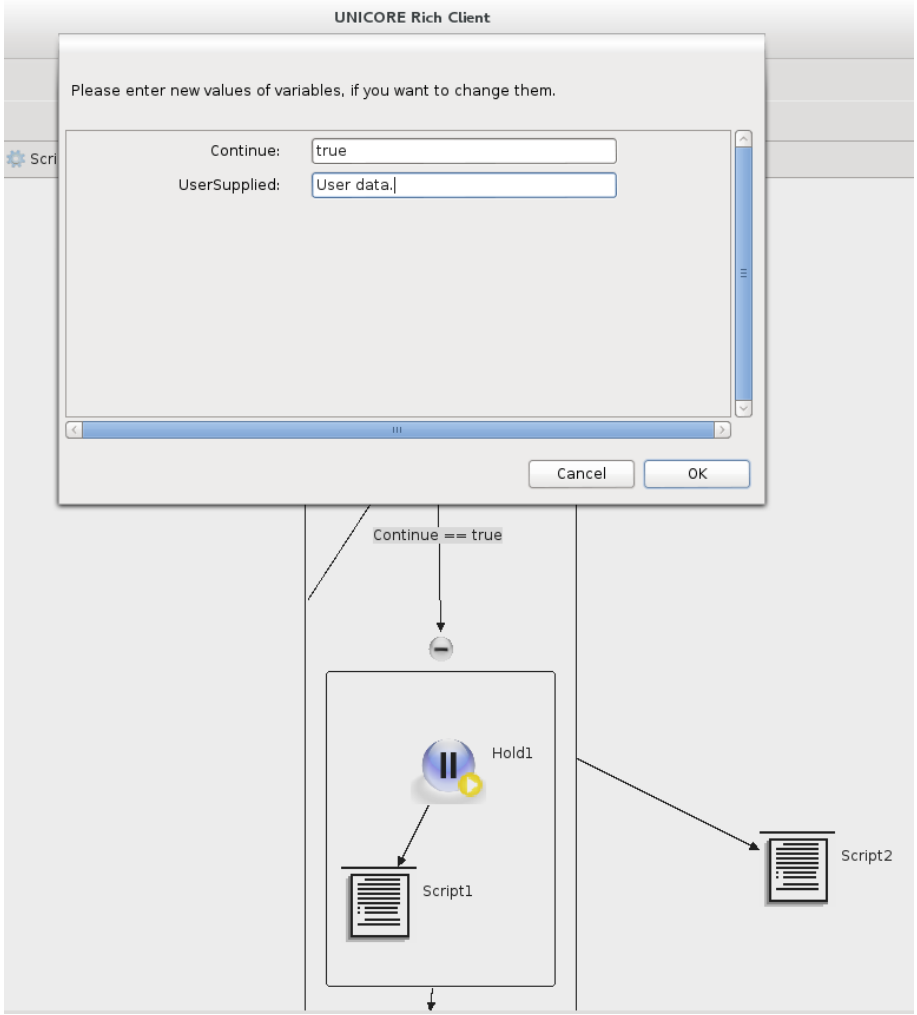

Figure 6. Modifying values of workflow variables at runtime.

variables. However, the URC did not provide a means to update and pass new values for workflow variables when resuming a workflow. This has been fixed by release 7.0.0 of the URC in May 2014.[7]

When a user wants to resume a workflow that is in the "hold" state, the URC displays a dialog listing all declared workflow variables of the workflow along with their respective current values. All of the values can be edited and updated before sending the actual resume request to the workflow engine. Figure 6 shows an example of the update dialog for a small workflow with only two variables. The hold statement of the workflow is embedded in a while loop that terminates once the value of workflow variable "Continue" is not "true" anymore. Thus, the user can define the final loop of the tasks within the while loop.

In older versions of the workflow engine and URC that already supported hold and resume, this behaviour can be mimicked by holding the workflow, modifying a file that contains the input parameters and resuming the workflow afterwards. However, this would only allow to indirectly influence the workflow control structures as no workflow variables can be modified in this way. Whereas for simple decisions it will be easy to make the control structures depend on the existence of files, which in turn depend on the provided values, numerical value or string comparison will pose problems for the older versions.

3.6 Parameterization

UNICORE workflows can be parameterized by introducing a number of workflow variables, the values of which control the execution of the workflow during runtime. An example of this, specific for the PLI workflow, is depicted in Figure 3.

3.7 Workflow Portability

One of the particular strengths of UNICORE is the simple transferability of complex workflows from one machine to another. Different target systems for the individual workflow applications can be easily selected in the resources panel of the UNICORE Rich Client. Concerning job imports and exports the user can rely on the UNICORE concept of a central workflow storage which manages automatically the data transfer between selected UNICORE storages with the best available protocol. Alternatively, if shared file systems are available the user can configure specially designed transfer job groups in the PLI workflow for controlling the transmission of data between different machines. The source of the raw data can be changed in a workflow variable as described in section 3.6. Finally, the workflow can be simply exported from the UNICORE Rich Client, so it can be exchanged between different end-users.

4 Results

Implementing this workflow using the UNICORE workflow engine resulted in reduced overall execution time. The actual figures are difficult to provide, as the implementations used in the workflow were parallel versions that were implemented shortly before the introduction of the workflow engine. Certainly, data movement and handling as well as manual invocation of the tools was avoided.

The UNICORE workflow system as a whole benefited from this complex use case in that a new feature has been added to the URC for modifying workflow variable when resuming held workflows.

In order to decrease the overall makespan of the workflow even further, we propose to run so called housekeeping jobs outside the batch system to avoid queuing time.

5 Future Work

During the preparation of the workflow, and even before instantiating it at different sites, we noticed that some information needs to be copied manually within the URC. For instance, storage URLs and mount points are required in several places of the workflow.

They can easily be injected into jobs through workflow variables. However, initially the information needs to be extracted from the Grid browser's details view and copied into these variables. We consider it to be a desirable feature to extract such information from the environment in the workflow editor.

Additionally, we found a need for the possibility to control workflow variable values from within jobs. This is not part of the UNICORE workflow system yet. Currently, the only information that the workflow engine can gather from within jobs are the existence and size of files as well as the exit codes of jobs[6]. An idea for implementing this would be to have defined job variable exports, analogous to file exports, that can be used as input to subsequent jobs and control structures of the workflow.

The version of the workflow described in this work is not final. There is still some potential for improvement, particularly to improve the maintainability of the housekeeping jobs even more by reducing them to the minimum. Our goal has been to document a complex workflow in order to derive general patterns that will be useful in subsequent work. The work on this workflow already resulted in additional features and optimizations in the UNICORE workflow system and URC.

We will continue to support complex workflows such as the PLI one in order to better understand user needs and integrate the requirements of such use cases as new features into UNICORE and its workflow system.

Finally, we would like to thank the Institute for Neuroscience and Medicine at Forschungszentrum Jülich GmbH for providing this interesting opportunity to incorporate their tools in a complex UNICORE workflow.

References

1. Anna Westhoff. Hybrid Parallelization of a Seeded Region Growing Segmentation of Brain Images for a GPU Cluster. In *ARCS 2014: 27th International Conference on Architecture of Computing Systems - Workshop Proceedings*, page 8, Berlin, 02/25/2014 - 02/28/2014 2014. 27th International Conference on Architecture of Computing Systems, Lübeck(Germany), VDE Verlag. ISBN 978-3-8007-3579-2.
2. Anna M. Westhoff. GPU-Accelerated Segmentation of High-Resolution Human Brain Images Acquired with Polarized Light Imaging. Ms, FH Aachen-Jülich, Jülich, 2013. URL `http://juser.fz-juelich.de/record/138037`. FH Aachen-Jülich, Masterarbeit, 2013.
3. Hubertus Axer, Markus Axer, Timo Krings, and Diedrich Graf v. Keyserlingk. Quantitative Estimation Of 3-d Fiber Course in Gross Histological Sections of the Human Brain Using Polarized Light. *Journal of Neuroscience Methods*, 105(2):121 – 131, 2001. ISSN 0165-0270. doi: http://dx.doi.org/10.1016/S0165-0270(00)00349-6. URL `http://www.sciencedirect.com/science/article/pii/S0165027000003496`.
4. M. Axer, K. Amunts, D. Grässel, C. Palm, J. Dammers, H. Axer, U. Pietrzyk, and K. Zilles. A Novel Approach to the Human Connectome: Ultra-High Resolution Mapping of Fiber Tracts in the Brain. *NeuroImage*, 54:1091 – 1101, 2011. ISSN 1053-8119. doi: 10.1016/j.neuroimage.2010.08.075.

5. J. Dammers, M. Axer, D. Gräßel, C. Palm, K. Zilles, K. Amunts, and U. Pietrzyk. Signal Enhancement in Polarized Light Imaging by Means of Independent Component Analysis. *NeuroImage*, 49:1241 – 1248, 2010. ISSN 1053-8119. doi: 10.1016/j.neuroimage.2009.08.059.
6. *UNICORE Workflow System Manual*, February 2014. URL `http://www.unicore.eu/documentation/manuals/unicore6/files/workflow/workflow-manual.html`. Last accessed: 2014-07-28.
7. *UNICORE Rich Client user manual*, July 2014. URL `http://www.unicore.eu/documentation/manuals/unicore6/files/urc/manual.html`. Last accessed: 2014-08-01.
8. Bastian Demuth, Lara Flörke, Björn Hagemeier, Michael Rambadt, Daniel Mallmann, Mathilde Romberg, Rajveer Saini, and Bernd Schuller. UNICORE Rich Client User Manual. In *Multiscale Modelling Methods for Applications in Materials Science*, volume 19 of *IAS Series*, pages 295–319, Jülich, 09/16/2013 - 09/20/2013 2013. CECAM Tutorial Multiscale Modelling Methods for Applications in Materials Science, Jülich(Germany), Forschungszentrum Jülich GmbH, Zentralbibliothek, Verlag.

Fostering the Adoption of UNICORE Portal

Krzysztof Benedyczak[1], Piotr Bała[1], Marcelina Borcz[1], Valentina Huber[2], Rafał Kluszczyński[1], Mariya Petrova[2], Piotr Piernik[1], and Bernd Schuller[2]

[1] ICM, University of Warsaw *E-mail: {golbi, bala, mborcz, r.kluszczyński}@icm.edu.pl, piotrpiernik@gmail.com*

[2] Jülich Supercomputing Centre *E-mail: {v.huber, m.petrova, b.schuller}@fz-juelich.de*

The UNICORE Portal is a new offering in the UNICORE client side tools portfolio. In this paper we are evaluating the current architecture of the Portal and its connection to the corresponding UNICORE middleware components from the perspective of domain-specific interfaces integration. The paper discusses the need for a low-level Portal API, which should serve as a convenient foundation of Portal extensions. The motivation of the developments is built on top of requirements coming from three use cases, which are introduced in the paper. The proposed API is compared to the GridBeans API, used for the standalone graphical clients. Additionally an analysis of the possible improvements in the Portal core is given, together with the description of the benefits of refactoring.

1 Introduction

The UNICORE Portal is a recently added element to the UNICORE portfolio of client applications. At the same time it is much different from the other, already existing client software, as it is using two separate software components to realize the client's functionality: a thin web-browser client (UNICORE unaware) and the Portal servlet which serves as a Grid gateway.

This design, typical in the Web world, brings several challenges: the Portal application is used by multiple users simultaneously what requires proper context separation and careful management of resources (disk space, memory) which are much more constrained than in the standalone client case.

The UNICORE Portal provides a ready to be used interface for generic jobs, which is a web version of the well-known Generic GridBean available in the UNICORE Rich Client. Still it can be assumed that the generic interface can be considered only as an additional tool, complementary to domain specific interfaces, tuned for particular use cases. In this work we provide three such use cases coming from the Polish Grid infrastructure, and developed in the PL-Grid Plus project[1].

In this paper we are evaluating the current architecture of the Portal and its connection to the corresponding UNICORE middleware components from the perspective of integration of the domain specific interfaces. We propose a low level Portal API, to form a foundation for a development of Portal extensions. Analogy to the GridBeans API, used for the standalone UNICORE graphical clients, is natural, therefore we compare both approaches. The Portal API alone, is not enough to improve Portal up take in our opinion. Stability and high performance are additional important vectors along which the Portal should be improved. Therefore we enumerate several possible improvements in the Portal core module.

Subsequently the paper lists the required improvements in the area of reusable components of the user interface. We provide a design of a flexible framework which can be used to easily develop new application specific interfaces, without restricting their functionality.

The final part of the paper covers several aspects of the improvements of the generic UNICORE middleware, which are needed to provide better scalability and performance of the UNICORE access from the web browser via a single portal entry point.

2 Portal Use Cases

The most important reason behind creation of the UNICORE Portal client was the end-user friendliness. The web-based client does not require installation (and updates) of specialized software on a user's machine, can be used from arbitrary locations and authentication is greatly simplified. Therefore it is not a surprise that the domain-specific users expect solutions which are bespoke exactly for their specialized needs.

The PL-Grid Plus project integrates several such domain specific solutions with UNICORE Grid middleware. The domain-specific interfaces are a very important part of the Portal as provide a much better experience then this provided by generic application interfaces: can handle customized output visualisation, limit and tune the widgets, provide advanced resource selection logic tailored to the application being used or use several Grid applications together with a simple to use common interface. One of those applications is a computed tomography (CT) image analysis program called SinusMed. SinusMed is a result of ICM[a] development in cooperation with medical doctors. The application takes a series of CT images as input, which as a whole constitute a 3D image of human head. The application recognizes all air-filled head areas, marks them and classifies by providing a mapping to the reference areas from a data set manually processed by medical personnel. The results are to be presented in the portal in a visual form as well as the user should be able to download them for visualization with standalone medical software.

Another use case comes from materials science. The Vienna Ab initio Simulation Package (VASP)[2] is used for atomic-scale materials modelling. It computes an approximate solution of the many-body Schrödinger equation. While the application is provided externally, the integration with UNICORE requires a dedicated interface allowing to easily prepare the input and manage simulations.

The final use case lies on the border of medicine and genomics. Using a complex UNICORE workflow invoking several existing applications, computations are performed to determine differences between tumour and non-tumour genome sequences of tissues obtained from patients diagnosed with colorectal cancer. Internally the computations require executing a complex workflow, dependent on particular simulation requirements set by the user. It is assumed though that the user should be only faced with his or her scientific parameters: the actual workflow structure must be built automatically basing on those requirements.

After the collection of a detailed set of functional requirements from the above use cases the following list of technical requirements was assembled:

[a]Interdisciplinary Centre for Mathematical and Computational Modelling (ICM) is a scientific division of the University of Warsaw.

- While the domain-specific interface in some cases can fit into a common user interface (UI) design, it is required to have full freedom in its design. This was especially requested by the VASP use case.
- The domain specific component must be able to produce and submit both atomic and workflow jobs created by a programmer with a use of a provided API.
- Several UI components are commonly needed by multiple domain specific modules, especially a flexible component for managing jobs table and a component monitoring the running job. We give further details in Section 5.
- Possibility to use all advanced features of the UNICORE middleware is required. Especially submission to the Service Orchestrator and access to the job's directory during the job's execution are important.

3 Overview of the Current Portal Architecture

The current architecture of the UNICORE Portal (as of version 1.0.0) is based on a custom solution for internal object management. The solution is configured with XML, where objects are defined. The XML is packaged within binary archives of the portal. The engine initializes singleton objects basing on the configuration and places them in a registry which can be accessed from arbitrary part of the application code. Examples of such shared objects include localized message providers, Grid applications registry, and many other internal objects. The configuration allows for registering arbitrary objects and also a range of predefined types of objects which are handled in a special way.

The most important example of special objects are user interface components which are also defined in the same XML configuration. The XML also allows for controlling the composition of the user interface. This feature can also be tuned by the system administrator using a configuration file, where particular components can be disabled. This allows for a partial portal customisation.

The complete Grid view is discovered and modelled on the Portal side. This is done separately for each user. The Grid view is retrieved in the polling fashion, i.e. the Portal queries top-level Grid nodes starting from UNICORE Registry, and then follows down to the components logically below the parent, finishing at individual job nodes. This state is cached and refreshed periodically.

The Grid model implementation was copied from the UNICORE Rich Client code with many modifications to meet the requirements of a multi-user environment. As the code was not cleaned after the import, there are situations where parts of the present code are actually not used at runtime.

The file IO of the Portal (including in the first place the access to users' workspaces kept on the Portal) is not performed directly but using the additional layer of the Apache Virtual File System. This design allows for using not only the local server's filesystem but also a remote UNICORE storage as a physical location of workspaces.

Last but not least the Portal code includes a range of user interface components. We can enumerate the most important ones: the Grid Browser, the Sites Table, Data Manager, and the Jobs Table. The user can create a job with help of the Generic Job component. This component is a close relative of the Generic GridBean available in the UNICORE

Module	NCSS	% of total	Notes
Applet integration	1316	3%	
Authentication	2382	6%	
User interface	9091	23%	
Workflow support	10809	27%	UI & logic
Core logic	16347	47%	In this grid model: 13%
TOTAL	**39945**		

Table 1. Code proportions between UNICORE Portal modules. NCSS stands for one of the standard code size metrics: Non Commenting Source Statements[3]. The large portion of the logic part is a potentially bad symptom. It can result from the URC client copy where parts of the code are simply unused at runtime and some parts are over engineered or wrapped to match different environment.

Rich Client: it is possible to fill the values of the job parameters as specified in the server's IDB, provide UNICORE resource requirements and to control file imports and exports.

The portal access is protected with a separated authentication module. The authentication module defines a plugin interface so that different authentication methods can be implemented. So far the SAML 2, X.509 in-browser certificate, user name and password, Kerberos and demo authentication methods are implemented.

Table 1 provides general statistics of the high level logical modules of the portal. In this paper we do not consider the workflow support part, which is still under development, but we provide its statistics in the table for completeness.

4 The Lessons Learnt from the GridBeans Approach

The GridBeans[5] model was introduced in Grid Programming Environment[4], at the beginning of the service oriented epoch of UNICORE as a universal client-side Grid application integration layer. Initially it was implemented for the UnicoreGS server, which was later abandoned. Subsequently the UNICORE 6 & 7 implementations which made their way into production, become a target environment.

The GridBean model provides an abstract view of the Grid, which is fully UNICORE independent. The goal was to support Globus Toolkit 4 and potentially also other SOA Grid middleware. In practice GridBeans have been used solely with UNICORE server side so far.

The Grid application abstraction in the GridBean was done with several assumptions in mind:

- The GridBean developer should work with a single Grid application only.
- The GridBean implementation must provide a user interface for preparing job input and optionally a user interface for output visualization.
- The fundamental GridBean task is to prepare the job description, based on the user-entered input.
- The GridBean should be instantiated and run in a hosting environment providing job submission and control features, as well as the cross-cutting interfaces useful for

all GridBean implementations: Grid resource requirements selection, environmental variables control and file staging management.

The realization of the GridBean idea has resulted in several hosting client applications. Nowadays there is only one used: UNICORE Rich Client (URC). Therefore the role of the very abstract and universal programming model of GridBeans is now reduced to a well-defined, but complex extension API of the URC. GridBeans are used quite popularly and a few scientific applications have their dedicated GridBeans, making the use of UNICORE convenient. For unsupported applications there is a special *Generic GridBean*, which supports arbitrary applications, with user interface built dynamically from the application description retrieved from a server.

Despite of the partial success of the GridBeans approach, there is a couple of drawbacks. First of all the GridBeans API is very complex. The complexity results from the targeting goal of being universal and utterly generic which has been to a large degree never achieved. Second, the GridBean API does not allow for creation of advanced interfaces: using multiple Grid applications, submitting UNICORE workflows instead of atomic jobs, performing job-specific steering (e.g. cancellation of the job upon certain runtime conditions). Finally the generic wrapping of each and every GridBean instance, which is given by the hosting environment, cannot be influenced by the GridBean. This results in overcomplicated user interfaces and reduced usability. For instance for many applications only one, two or even none of Grid resource requirements make any sense. At the same time all resource requirements are presented to the user always.

The reason of those limitations is the *closed API* that is facilitated: a GridBean implementation must implement a given interface, returning a fixed set of objects to the environment. This approach can be put in opposition to an *open API* where each plugin implementation can use services and components provided by the API while their choice, configuration and composition is left to the plugin implementation. It can be also noted that while the closed API can be always extended with additional methods to configure the hosting environment, such change is required for each feature not foreseen in the API and required by a new plugin. With the open API even a missing feature can be directly implemented in the plugin, simply not relying on a missing functionality of the hosting environment.

5 Proposed Roadmap for the Portal

The main goal of our work was to analyse the Portal from the perspective of using it as a foundation for integration of domain specific interfaces, examples of which were given in Section 2. We came to several conclusions, some of which should have positive impact also for the Portal as seen as a generic UNICORE client solution.

Our first observation is about the Grid state discovery and modelling. As it was noted the portal is polling the Grid for the updates. As the portal is accessed by many users this "maintenance" action consumes a large amount of resources (there is refresh of each node per user), many refresh queries are superfluous (when there is no change) and finally the user is presented with the updated state of resources only after some time. To overcome all of those issues we propose to add an event system to UNICORE, where the servers push notifications to a broker whenever the state of a resource interesting from the client

perspective (job, target system) changes. Each client, including the Portal could then register to the message broker to be notified immediately about a change, also with a minimal overhead.

In the ideal world events in such a system should provide all necessary update information. However, as this imposes a lot of security challenges (access to unauthorized information) we propose to simplify the system, so that only notifications about a fact whose state has changed are sent. Then a client still has to poll for the updated resource state, but the polling will take place only once per change and will be performed immediately after it happens.

The push-approach should solve the problem with the Portal's access to the Grid state. However, the access to the Grid model inside the Portal requires simplification on its own. The current model is based on the URC model used for the Grid browser view. Changes in the model trigger notifications, the events are fine-grained and again Grid browser focused (for instance there is an event sent when node's icon is changed). In principle the model is sufficient, although it should:

- Produce events which are useful for the remaining portal components, i.e. which are not bound to the physical Grid layout. For instance it is not possible to register a listener for arbitrary job status changes. Instead there are only events on children changes of a particular parent node that is interesting for the Grid browser only.

- The whole model should be cleaned up, so that unused code imported from URC is removed. The need for this simplification is also resulting from the analysis of the amount of business logic related code as shown in Table 1.

The second change proposed for the portal is the introduction of a portal internal API. The API should be documented and available in a single package. Features of such an API should include:

- Complete Grid state discovery. Naturally the implementation should be based on the aforementioned Grid model. Features must include job discovery, service discovery, Grid resources catalogue as offered by services as well as the available applications.

- Access to job submission service, with a simple interface allowing to simply submit jobs through UNICORE broker or workflow service (depending on job type).

On top of such a high-level Portal API the development of domain specific interfaces business logic will be simplified. To further help the developers, a range of reusable UI components is needed. This is potentially the smallest change from those proposed: most of the required components already exist. However, those components are not meant to be used as generic, embeddable components. The list of the most useful components follows:

- Job table viewer. The component should allow for programmatic control of contents (job filters, e.g. to show only jobs of a particular application), and the control of displayed columns. The component should not require manual refresh - the contents must be kept in sync with the portal representation of the Grid automatically. Whether the same or a separate component is available for displaying workflows is not relevant, assuming that such functionality exists.

- Resource requirements selection component. The component should allow for controlling the displayed resources pool.

- File monitoring component. The component should monitor a given file, displaying its contents. A possibility to provide a custom visualization implementation is needed.

- Components to easily upload and download files to/from the Grid storage.

- Components useful for job preparation: job staging control and variables control. Both are available as a part of the generic job preparation UI.

- A simple component monitoring a given job status. The component should also allow for aborting the monitored job.

Finally we suggest to base the portal on the well-established IoC framework[6]. This should make the initialization order of the portal components and the intra-dependencies management cleaner than it is now. Usage of an external IoC framework will allow also for removing parts of the code used currently in the portal. The XML configuration still can be used but only to define the composition of UI components displayed in the Portal.

6 Summary

In this position paper we have analysed the current UNICORE Portal architecture from the perspective of domain specific interfaces development. We propose to create a portal API allowing for greatly simplified development of such interfaces, along with a series of improvements in the Portal core. The proposed API is different from the formerly used URC GridBeans API in the sense that it is fully open: the user can exploit the features of the API freely, while creating interfaces without any API-imposed restrictions. At the same time, we resign from the GridBeans model abstraction which turned out to be useless in practice.

While we consider such an approach the best foundation for fostering the Portal adoption for domain-specific environments, we still accept the closed API approach, similar to the (simplified) GridBeans approach. Such restricted API can simplify development of "standard" interfaces of well-known applications executed in a batch mode. Therefore it should be considered as an additional, option: simpler to develop but limited in flexibility offered to developers.

A somehow negative consequence of UNICORE Portal development is the necessity to resign from the GridBeans which were developed so far. A translation layer is theoretically possible, but would require to first of all implement GPE API in the Portal (what is considered a bad approach on its own as described above) and then a giant bridge for the UI library used by GridBeans (which is Eclipse SWT). As such development is clearly larger than a re-implementation of all available GridBeans from scratch it should not be considered. However we can argue that this situation is not as bad as looks at the first sight. The GridBeans UI is far from the interfaces of the Web. Therefore more customised Web-oriented application interfaces might suit users more than the fixed GridBeans layout.

7 Acknowledgements

This work was made possible with assistance of the PLGrid Plus project, contract number: POIG.02.03.00-00-096/10, website: www.plgrid.pl The project is co-funded by the European Regional Development Fund as part of the Innovative Economy program.

References

1. The PlGrid Plus project: http://www.plgrid.pl/en (28.07.2014)
2. Vienna Ab initio Simulation Package: http://www.vasp.at/ (28.07.2014)
3. Non Commenting Source Statements definition: http://www.kclee.de/clemens/java/javancss/ (26.07.2014)
4. The Grid Programming Environment project: http://gpe4gtk.sourceforge.net/ (26.07.2014)
5. Sandra Bergmann: GridBean Developer's Guide. http://www.unicore.eu/documentation/manuals/unicore6/files/GridbeanDevelopersGuide.pdf (26.07.2014)
6. Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern. 23 January 2004, https://blog.itu.dk/MMAD-F2013/files/2013/02/3-inversion-of-control-containers-and-the-dependency-injection-pattern.pdf (28.07.2014)

Certificate-free User-friendly HPC Access with UNICORE

Richard Grunzke, Ralph Müller-Pfefferkorn

Center for Information Services and High Performance Computing
Department of Distributed and Data intensive computing
Technische Universität Dresden
Dresden, Germany

UNICORE as a middleware to uniformly access computing resources is currently hampered in its daily use by its requirement of using personal certificates. For users these are inconvenient and complex to handle. The latest major release of UNICORE enables its certificate-free deployment using Unity as translating service for external authentication sources such as LDAP. The paper describes the background, the approach taken, experiences and gives an outlook.

1 Introduction

Currently, the use of UNICORE and its further adoption is significantly hampered by the fact that grid certificates are required by users. These have to be manually applied for at official registration authorities where an official ID has to be presented in person. Furthermore, this procedure has to be repeated every year in order to prolong the certificate. Especially for novice users this is a significant barrier of entry and annoyance during usage. This requirement prevents UNICORE from becoming a truly universal and usable access to HPC resources.
With the release of UNICORE version 7.0.0 and Unity 1.0.0 in January 2014 an integrated solution path becomes available that enables users to simply use their, for example, university identities to access HPC resources via UNICORE. Unity is an independent solution that enables to interface with a diverse set of federated identity management systems and provides authentication and authorisation functions in a graphical way. Unity can be configured to serve in a diverse set of use case scenarios. One being that it can enable UNICORE to truly become a standard way of access for complex HPC resources. Even novice users are enabled to immediately use these in order to easily run highly complex computing tasks.
We are working towards our goal at ZIH to automatically enable all HPC users to use UNICORE clients for HPC access without any further step needed on the users' sides. Besides the available UNICORE rich and commandline clients, the UNICORE portal is being deployed at ZIH and will be integrated with Unity and LDAP to make HPC access even easier. We see the integration of the ZIH LDAP installation with UNICORE as a significant lowering of the hurdle of use to ZIH HPC resources.

2 Background and Related Work

UNICORE is a universal HPC middleware used in university computing centres and large research infrastructures such as XSEDE[1], PRACE[2], EGI[3] and soon the Human Brain

Project[4]. It enables the uniform integration of heterogeneous computing and data infrastructures. Access is possible via the platform independent clients such as the UNICORE portal[5], rich[6] and commandline client. The workflow and metadata management[7] provide higher level of services to manage computing tasks and data sets. Furthermore, concepts such as the data oriented processing[8] and the space based approach[9] enable integrated and efficient support for high throughput data life cycles. Furthermore, fully-fledged virtual research environments such as the MoSGrid science gateway[10] integrated UNICORE and make use of its HPC, data and metadata[11] functionality.
According to[12] "Unity is a complete solution for identity, federation and inter-federation management. Or, looking from a different perspective, it is an extremely flexible authentication service. Authentication of web and cloud resources can be outsourced to Unity. Unity can act as a bridge to a not cloud/web friendly users' store. Finally Unity can help to integrate 3rd party identity providers with local resources." It is used by the Polish national Grid infrastructure and will be used by the EU Flagship Human Brain Project[4].
LDAP[13] (Lightweight Directory Access Protocol) is a widely adopted standard commonly used for distributing authentication and authorization information. One such an example is exposing central information about users in a controlled manner for external services. By making use of this user information these services can, for example, enable users to log in via credentials they already have.

3 Approach

Figure 1 depicts the general architecture in relation to the topic at hand. Central LDAP identity management service exposes login information for certain controlled and activated services. This includes user names and passwords as well as group memberships. One such group indicates that a user is allowed to access a cluster at ZIH. UNICORE is used to facilitate the access to such clusters. UNICORE cannot directly make use of LDAP information. This challenge is met with using Unity as an identity translation service. A UNICORE client such as the commandline or Rich client can then be used to access the cluster. The client contacts Unity and presents its user name and password. Unity then transparently asks the LDAP server for confirmation. Then the user is allowed to access the cluster without use of a certificate. He just needs a standard keystore containing the public keys of relevant certificate authorities. Users that have access to a specific cluster via a granted resource application will automatically also have access via UNICORE to this specific resource. In the following details about the situation are described.

UNICORE/X Configuration

UNICORE/X has to be able to issue trust delegations in the name of the user. In order to do this it requires the public key which is used to configure the Unity service. This is configured in the wsrflite.xml configuration file of UNICORE/X.

```
<property name="container.security.trustedAssertionIssuers.
type" value="directory"/>
<property name="container.security.trustedAssertionIssuers.
directoryLocations.1" value="conf/unity/unity.pem"/>
```

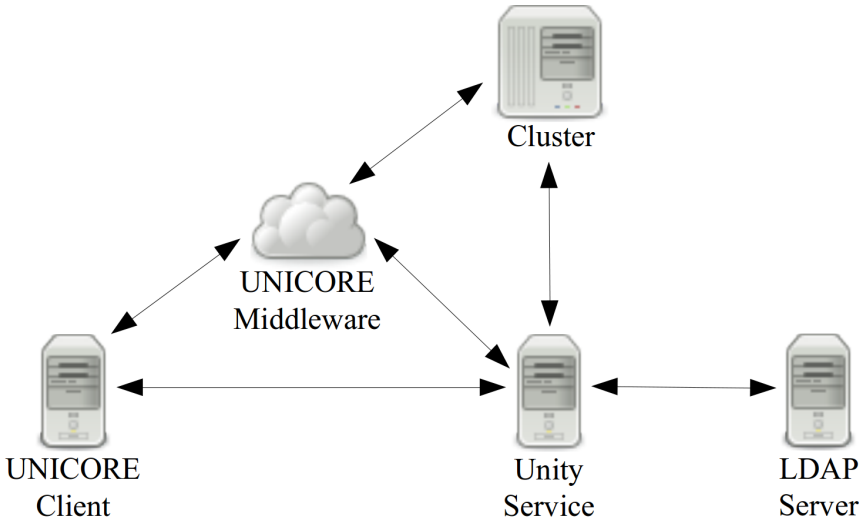

Figure 1. The LDAP Server is shown as the source of identity information. Unity enables users by a UNICORE client to access a cluster via UNICORE with their already existing logins. A client can be the commandline client, the rich client, or the portal.

In the uas.config of UNICORE/X an external authentication service has to be configured by defining a further configuration file containing the respective details.

```
container.security.attributes.VO-PULL.class=eu.unicore.uas.
security.vo.SAMLPullAuthoriser
container.security.attributes.VO-PULL.configurationFile=
conf/vo.config
```

Then, in the vo.config file the use of the Unity service is defined as authentication source.

```
vo.pull.enable=true
vo.pull.enableGenericAttributes=true
vo.pull.voHost=clavius.zih.tu-dresden.de
vo.pull.voPort=2443
```

UNICORE Gateway Configuration

In the UNICORE gateway the client authentication has to be turned off in the gateway.properties configuration file. This property disables the enforcement of SSL client certificate authentication. This is required since a client, such as the UNICORE Rich Client, cannot be counted on to have a private key as is the case when using Unity.

```
gateway.httpServer.requireClientAuthn=false
```

Unity Configuration

The following listing of unityServer.conf show the specific part in configuring Unity for LDAP[14] with configuring the endpoint for UCC and URC and the authenticator. An endpoint is the interface to applications that accesses Unity functionality. An authenticator can be used to authenticate clients by external means such as LDAP in this case. To be used by an endpoint an authenticator can be bound to it via the *endpointAuthenticators* parameter.

```
unityServer.core.authenticators.1.authenticatorName=ldapWeb
unityServer.core.authenticators.1.authenticatorType=
ldap with web-password
unityServer.core.authenticators.1.verificatorConfigurationFile=
conf/authenticators/ldap.properties
unityServer.core.authenticators.1.retrievalConfigurationFile=
conf/authenticators/passwordRetrieval.json

unityServer.core.endpoints.1.endpointType=
SAMLUnicoreSoapIdP
unityServer.core.endpoints.1.endpointConfigurationFile=
conf/endpoints/saml-webidp.properties
unityServer.core.endpoints.1.contextPath=
/unicore-soapidp
unityServer.core.endpoints.1.endpointRealm=defaultRealm
unityServer.core.endpoints.1.endpointName=
UNITY UNICORE SOAP SAML service
unityServer.core.endpoints.1.endpointAuthenticators=
ldapWeb
```

Here example content for the respective ldap.properties file to configure the LDAP authenticator is shown.

```
ldap.servers.1=ldap-test.zih.tu-dresden.de
ldap.ports.1=636
ldap.userDNTemplate=uid={USERNAME},ou=users,dc=tu-dresden,dc=de
ldap.translationProfile=ldapProfile
ldap.attributes.1=uid
ldap.groupsBaseName=dc=tu-dresden,dc=de
ldap.groups.1.objectClass=groups
ldap.groups.1.memberAttribute=memberUid
ldap.groups.1.matchByMemberAttribute=cn
ldap.groups.1.nameAttribute=cn
```

UNICORE Commandline Client Configuration

The following are the configuration options with using UCC with the example configuration at ZIH. The option *authenticationMethod* is used to indicate that Unity is used for authentication while *unity.address* specifies the Unity server address. Both are mandatory while *unity.username* and *unity.password* are optional and used to specify the central user name and password as exposed via LDAP.

```
authenticationMethod=unity
unity.address=https://clavius.zih.tu-dresden.de:2443/
unicore-soapidp/saml2unicoreidp-soap/AuthenticationService
unity.username=username
unity.password=password
```

UNICORE Rich Client Configuration

In Figure 2 the configuration for using Unity within the UNICORE Rich Client is displayed. When the URC is started the user chooses *Unity* and enters the Unity server address, the user name and password as stored in LDAP.

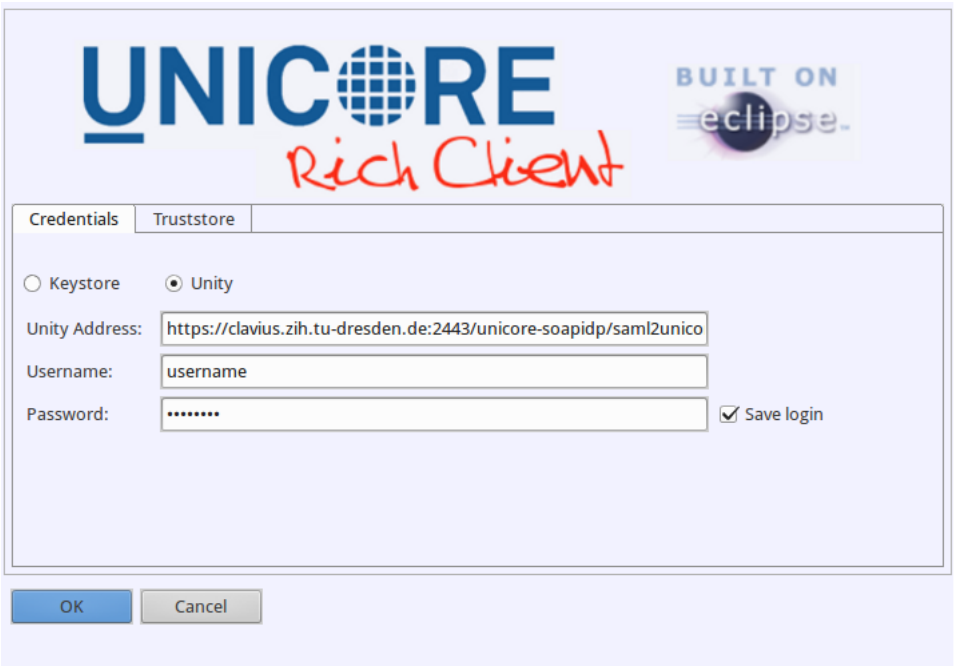

Figure 2. UNICORE Rich Client Configuration using Unity

Under the *Truststore* tab the user can specify a JKS keystore, a directory with certificates or an OpenSSL public key directory. Figure 3 displays the configuration when a JKS keystore is used.

4 Experiences

The experiences are quite positive. UNICORE and Unity are able to handle such a complex situation due to their flexible nature. Due to the complexity it takes time to deeply grasp the

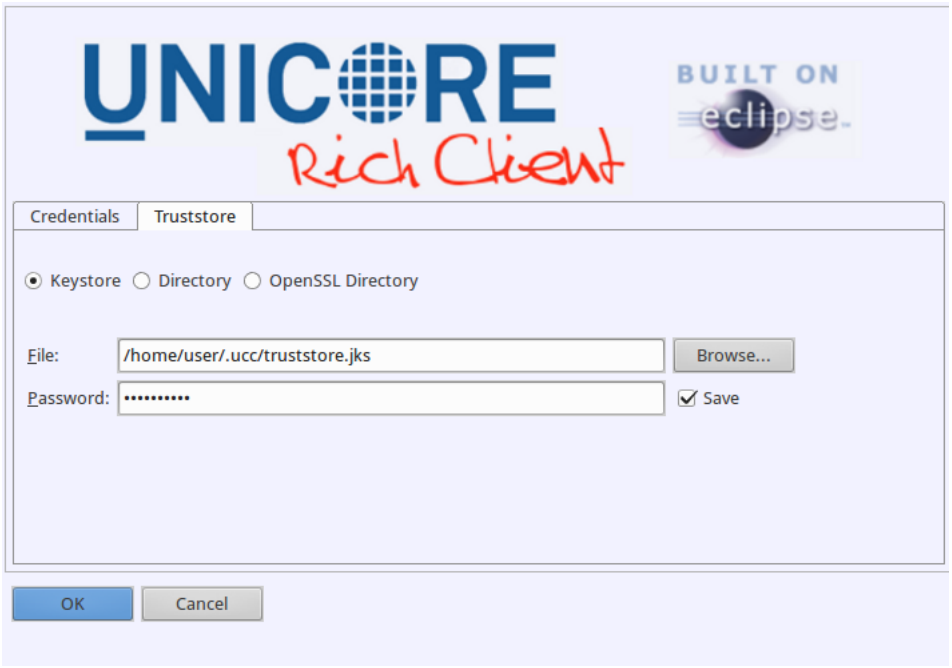

Figure 3. UNICORE Rich Client Configuration using Unity

situation, the capabilities of the involved systems and how they can be integrated to best meet the needs of the users. The documentation was generally good and the support is fast and excellence, as always. It is expected that this deployment will establish UNICORE as a standard service for access to HPC systems by daily users in easy, uniform and standard way.

5 Summary and Outlook

The paper describes the situation of UNICORE in a complex environment especially in regard to users and their usability needs. A path is described that relieves users of handling personal certificates for authentication while at the same time enabling them to access arbitrary computing resources via UNICORE. LDAP as identity backend and Unity as translation services are integrated with UNICORE to accomplish this. Besides the bigger picture configuration details are described.
Further steps are the integration with DFN-AAI via Shibboleth to also enable access for German users not affiliated with ZIH. Also planned is the integration with research environments such as the MoSGrid science gateway to enable these users a certificate-free access to HPC resources.

Acknowledgements

The authors would like to thank the Helmholtz Association of German Research Centres for the opportunity to do research in the LSDMA project. The research leading to these results has also partially been supported by the European Commission's Seventh Framework Programme under grant agreement no 312579 (ER-flow). Especially Bernd Schuller and Krzysztof Benedyczak shall be thanked for the fruitful discussions.

References

1. XSEDE, "Extreme science and engineering discovery environment," 2014. [Online]. Available: https://www.xsede.org
2. PRACE, "Partnership for Advanced Computing in Europe," 2014. [Online]. Available: http://www.prace-ri.eu
3. EGI, "European grid infrastructure," 2014. [Online]. Available: https://www.egi.eu
4. HBP, "The human brain project," 2014. [Online]. Available: https://www.humanbrainproject.eu
5. M. Petrova, V. Huber, B. Demuth, K. Benedyczak, and B. Schuller, "The UNICORE Portal," in *UNICORE Summit 2013 Proceedings*, IAS Series, vol. 21, 2013.
6. B. Demuth, B. Schuller, S. Holl, J. Daivandy, A. Giesler, V. Huber, and S. Sild, "The UNICORE Rich Client: Facilitating the Automated Execution of Scientific Workflows," in *e-Science (e-Science), 2010 IEEE Sixth International Conference on*. IEEE, 2010, pp. 238–245.
7. W. Noor and B. Schuller, "MMF: A flexible framework for metadata management in UNICORE," in *UNICORE Summit 2010 Proceedings*, IAS Series, vol. 5, 2010, pp. 51–60. [Online]. Available: http://hdl.handle.net/2128/3812
8. B. Schuller, R. Grunzke, and A. Giesler, "Data Oriented Processing in UNICORE," in *UNICORE Summit 2013 Proceedings*, IAS Series, vol. 21, 2013, pp. 1–6.
9. R. Grunzke and B. Schuller, "Secure High-Throughput Computing Using UNICORE XML Spaces," in *UNICORE Summit 2010 Proceedings*, IAS Series, vol. 5, 2010, pp. 27–35. [Online]. Available: http://hdl.handle.net/2128/3812
10. J. Krüger, R. Grunzke, S. Gesing, S. Breuers, A. Brinkmann, L. de la Garza, O. Kohlbacher, M. Kruse, W. E. Nagel, L. Packschies, R. Müller-Pfefferkorn, P. Schäfer, C. Schärfe, T. Steinke, T. Schlemmer, K. D. Warzecha, A. Zink, and S. Herres-Pawlis, "The mosgrid science gateway – a complete solution for molecular simulations," *Journal of Chemical Theory and Computation*, vol. 0, no. ja, p. null, 2014. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/ct500159h
11. R. Grunzke, S. Breuers, S. Gesing, S. Herres-Pawlis, M. Kruse, D. Blunk, L. de la Garza, L. Packschies, P. Schäfer, C. Schärfe, T. Schlemmer, T. Steinke, B. Schuller, R. Müller-Pfefferkorn, R. Jäkel, W. E. Nagel, M. Atkinson, and J. Krüger, "Standards-based Metadata Management for Molecular Simulations," *Concurrency and Computation: Practice and Experience*, 2013. [Online]. Available: http://dx.doi.org/10.1002/cpe.3116

12. Unity, "Unity - cloud identity and federation management," 2014. [Online]. Available: http://unity-idm.eu
13. T. A. Howes, M. C. Smith, and G. S. Good, *Understanding and deploying LDAP directory services*. Addison-Wesley Longman Publishing Co., Inc., 2003.
14. Unity, "Unity 1.3 LDAP Configuration," 2014. [Online]. Available: http://www.unity-idm.eu/documentation/unity-1.3.0/manual.html#_ldap

Perspectives for RESTful Services in the UNICORE Services Environment

Bernd Schuller, Jedrzej Rybicki, and Krzysztof Benedyczak

b.schuller@fz-juelich.de, j.rybicki@fz-juelich.de, golbi@icm.edu.pl

This paper describes the architecture, design and current implementation status of RESTful services within the UNICORE Services Environment. Initial prototype implementations and first applications are presented. Special care is given to the design of a flexible security layer for RESTful services which is both lightweight, and compatible with the current UNICORE 7 services.

1 Introduction

UNICORE as an integrated federation software suite offers very powerful clients, both graphical and commandline, which have been successfully used in many usage scenarios. Still, there are cases that cannot easily be implemented using the existing UNICORE clients, and the need for lower-level access arises. Even using the Java APIs offered by the UNICORE client libraries is sometimes not enough, and direct use of the services is required. In its current version 7.0, UNICORE can be accessed through SOAP/WSRF web services[1], coupled with a security layer based on SSL, SAML, and XML digital signatures. All of these are open, well-documented standards, and in principle it would be possible to implement clients to access the middleware services in any language. However, practical experience has shown that due to the high complexity of SOAP/WSRF and SAML only Java and C# offer the required libraries.

Consequently, it can be difficult or even impossible to use the current Web Service APIs offered by UNICORE. For example, this may occur when integrating UNICORE services into existing applications or community workflows.

Thus, the need for simpler service APIs arises. A popular alternative to SOAP are RESTful services[2]. These are usually more lightweight and more easily accessible for a number of reasons. They typically make use of JSON[3] instead of XML for resource representations and exploit HTTP semantics for the resource manipulation instead of defining their own. To make implementation of clients easier, one would like to avoid handling digital signatures on the client, so more lightweight security mechanisms such as OpenID-Connect[4] are of considerable interest.

We consider two use cases. Firstly, in the European Human Brain Project[5], UNICORE will form the basis for the project's high-performance computing (HPC) Platform. It comprises of four major HPC sites, cloud storage and other resources. Here, developers want to write Python applications for accessing services of the HPC Platform such as job management or data transfer from a Python application. OpenID-Connect should be used as the authentication mechanism. The second use case is a standalone client for the UNICORE file transfer protocol (UFTP)[6], which allows users to access their data without requiring the use of a full UNICORE client. In both these use cases, strong authentication and delegation of rights are required, and it is important to stay compatible with the usual access through UNICORE.

The remainder of the paper is organised as follows. In the next section we describe the UNICORE Services Environment, which forms the foundation of a UNICORE server. In section 3 we present the extensions we have implemented for realising RESTful UNICORE services. Section 4 presents some first applications, and the paper concludes with a summary and outlook to the next development steps.

2 The UNICORE Services Environment

The UNICORE Services Environment (USE) is the set of software components that provides the ecosystem for UNICORE services.

USE is built around Apache CXF[7] as the underlying web services framework. This is a very mature and well-maintained open source product, that provides support for both SOAP and RESTful web services. It has a powerful and flexible API and could be embedded easily into the UNICORE environment. CXF supports the JAX-RS specification[8], which makes development of RESTful services very convenient.

Efforts have been made to decouple the front-end service implementation (e.g. SOAP WSRF) as much as possible from the internal state, aiming for the model-view-controller pattern.

In UNICORE 7 the security architecture has been enhanced by allowing more flexible authentication through the use of the new Unity service[9]. Most importantly, in UNICORE 7 it is no longer mandatory for users to possess X.509 certificates, and can login via Unity with username and password or any other mechanism supported by Unity. This is a crucial improvement, since a requirement for client certificates would make the REST clients and services much more complex.

Figure 1 shows an overview of the USE layers and components, and how RESTful services fit in. This includes configuration handling, the persistence subsystem, the web server (Jetty) and the security subsystem with authentication, authorisation and access control.

3 RESTful Services in USE

We have implemented a number of extensions to link the basic JAX-RS implementation provided by Apache CXF to the UNICORE resources framework provided by USE. These are contained in the module "use-rest" in the USE source code repository, which is available as open source at SourceForge[10].

- An authentication handler uses the HTTP basic authentication header (i. e. username and password) and maps them to a X.500 DN using a configurable chain of authentication components;
- Several authentication components have been implemented. Users can be authenticated using a local username/password file, or the admin can delegate the authentication to Unity. When using Unity, a trust delegation assertions is obtained automatically, which can be used to make delegated calls to other UNICORE services via the SOAP/WSRF interface;
- Mechanisms were implemented that inject the requested resources and other required information (such as the Kernel and Home classes) into the JAX-RS service class;

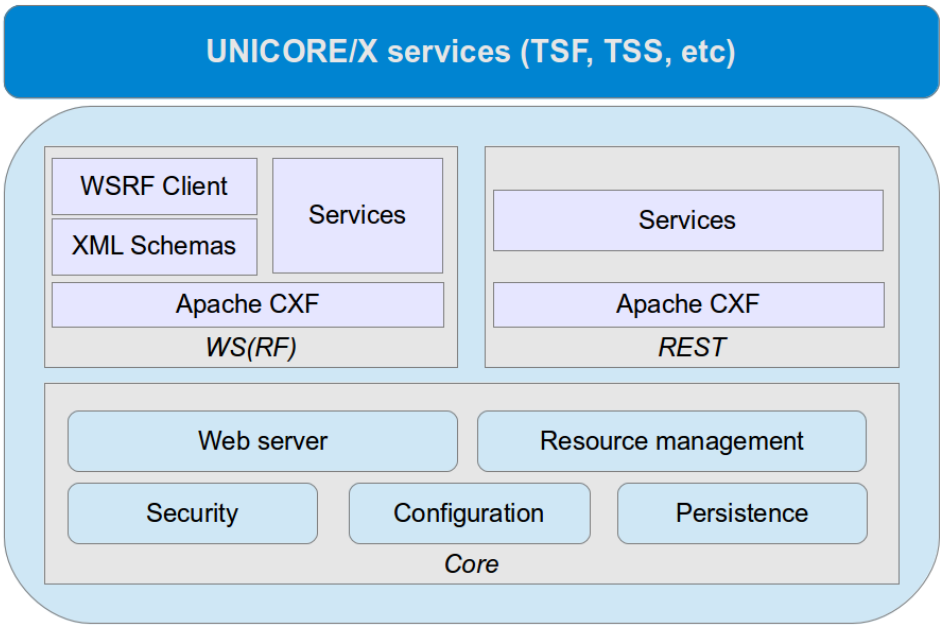

Figure 1. Overview of the USE architecture

- Access control checks are done using the usual XACML policy decision point;
- Though not part of USE, the UNICORE Gateway was enhanced to more fully support the HTTP methods GET, POST, DELETE, PUT and HEAD.

Authentication components are configured similarly to the attribute sources (e. g. the XUUDB) as a chain of authenticators, where each authenticator has the following interface:

```
public interface IAuthenticator {

  public void authenticate( HttpServletRequest request,
                            SecurityTokens tokens);

}
```

Authenticators can use information from the HTTP request, for example special HTTP headers, to make their decision. Successful authentication means that the security tokens will have a valid distinguished name (DN) for the current user.

Access control uses the well-known mechanism: the DN is mapped to user attributes by the container's attribute sources, then the XACML policy decision point is invoked to check permissions for accessing the current resource.

Summarizing, REST services need additional configuration in a UNICORE container:

- service definitions (wsrflite.xml file)
- authentication handlers
- policy entries to allow access to the services

Since a crucial requirement of this work is to have consistent access to UNICORE from both SOAP/WSRF and REST, we implemented extensions to the Apache CXF REST support which allow to inject the required model classes and related infrastructure classes like `Kernel` into the JAX-RS resource classes.

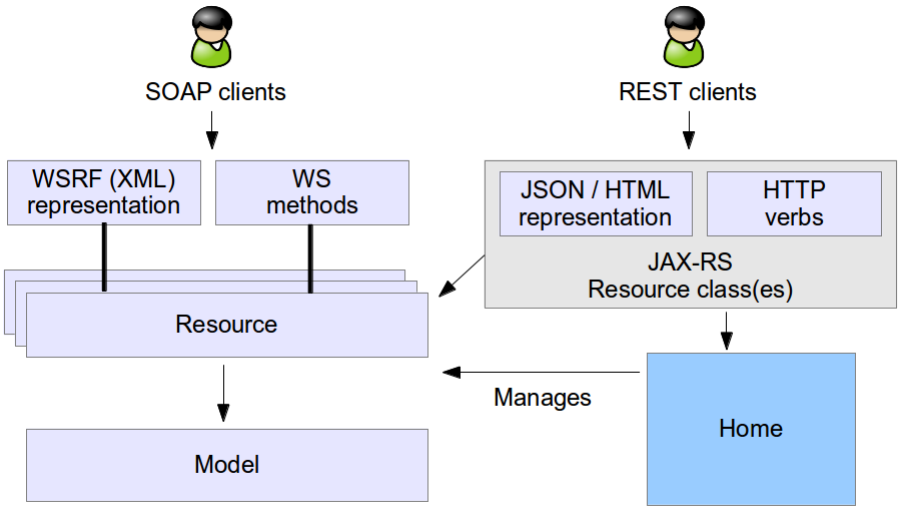

Figure 2. Resources and representations in USE

Figure 2 shows the basic relationships. The Home class is responsible for bookkeeping tasks, and controls the basic lifecycle of the resources. Each type of resource (e. g. jobs) has one associated Home class. Each resource has a model, which is persisted to disk. Representations are served to the clients (XML in the WSRF case, JSON or HTML in the REST case). The main motivation to use JSON instead of XML in the REST case is the greatly reduced overhead of JSON, both on the wire and in terms of processing. Also, from a programmer's perspective, producing and consuming JSON is often easier to implement compared to the equivalent XML.

Resources can be modified by invoking Web Service methods (SOAP/WSRF) or by using HTTP methods (POST, PUT, etc.) in the REST case.

For example, a job resource in UNICORE represents the actual job that was submitted by the user, and which is executed on the underlying batch system. The job resources' model is used to persistently store additional information, such as the job's owner, termination time and other data such as references to the parent target system or job directory

resources. When a client accesses the resource, it can either retrieve a representation or modify the resource, for example abort a job. In the SOAP/WSRF case, representations are defined via an XML Schema, and are called "resource properties" according to the WSRF specification. For illustration, a job resource properties document looks like this (heavily abbreviated):

```
<jms:JobProperties xmlns:jms="...">
   <jms:SubmissionTime>2014-07-23T11:29:15</jms:SubmissionTime>
   <jms:OriginalJSDL>...
   <jms:ExecutionJSDL>...
   ...
   <jms:Log>...</jms:Log>
   <jms:TargetSystemReference>...
   <jms:WorkingDirectoryReference>...
   <jms:Queue>batch</jms:Queue>
   <typ:StatusInfo xmlns:typ="...">
     <typ:Status>SUCCESSFUL</typ:Status>
     <typ:Description/>
     <typ:ExitCode>0</typ:ExitCode>
   </typ:StatusInfo>
   ...
</js:JobProperties>
```

In the REST case, multiple types of representations can be served, depending on the service's capabilities and the content type requested by the client using the HTTP "Accept" header. We have implemented base JAX-RS classes that serve JSON and HTML representations, and which can be extended easily to provide the actual service implementations.

For example, the JSON rendering of a job looks like this:

```
{
    "status": "SUCCESSFUL",
    "resourceStatus": "READY",
    "currentTime": "2014-07-23T14:16:34+0200",
    "terminationTime": "2014-08-22T14:03:42+0200",
    "queue": "N/A",
    "uniqueID": "bc5052cb-e300-45de-a2f6-051c493363db",
    "parentTSS": "bdc5fa10-e908-4f25-9667-f5aac2c556fc",
    "owner": "CN=Demo User,O=UNICORE,C=EU",
    "submissionTime": "2014-07-23T14:03:42+0200",
    "statusMessage": "",
    "exitCode": "0"
}
```

As is obvious, the JSON format is much less verbose than the XML, and consumes less resources for producing, transmitting over the wire and parsing on the client.

4 First Applications

As a first simple application, we have implemented a standalone authentication service for UFTP. This work was already ongoing, and we ported it to be a UNICORE REST service, to be able to leverage the existing authentication, attribute sources and access control infrastructure of the UNICORE Services Environment. This service will accept data transfer requests from authenticated and authorised users, forward them to the UFTPD server, and reply to the client with the address and port of the UFTPD server. The client can then launch the data transfer or other UFTP operation like file synchronization. While this UFTP authentication service is fairly trivial, it opens up the use of UFTP to a much wider user base. In many cases users simply need a simple and powerful data transfer facility, and forcing them to use a full UNICORE installation would be too much overhead. This code is available on SourceForge[11] and will be released separately.

As a more elaborate application, we have started to implement the basic UNICORE services as RESTful services. The initial approach is to have a RESTful resource for each of the basic UNICORE entities, *sites*, *storages* and *jobs*.

Each of these supports HTTP GET for retrieving a list of accessible instances, and a GET with appended resource ID to retrieve a detailed representation of the particular instance. The HTTP DELETE method is used to destroy an instance. The common functionality has been put into a generic base class named `BaseRESTController`.

Table 1 shows the current state of the API. This API will be further evolved, aiming to

Table 1. Initial REST API for UNICORE services.

HTTP method on resource	*Description*	*Media type*
GET /jobs	Lists jobs accessible to the user	JSON, HTML
POST /jobs	Submits a new job	JSON
GET /jobs/{id}	Get representation of a job	JSON, HTML
DELETE /jobs/{id}	Remove a job	
GET /storages	Lists all storages accessible to the user	JSON, HTML
DELETE /storages/{id}	Remove a storage	
GET /sites	Lists sites accessible to the user	JSON, HTML
DELETE /sites/{id}	Remove a site	

support also access to and modification of individual resource properties, e.g. termination time, or things like the job name, or security information like the VO membership.

Of particular interest is *job submission*, since this will be the first non-trivial functionality required by the application developers in the Human Brain Project (HBP). In principle, job submission can be done in several ways, and even multiple ways are conceivable. To keep the client workflow simple, the simplest option is to POST a JSON job description to the */jobs* resource. Certainly it should be possible to also allow XML here, since the JSDL standard used in UNICORE is well-known. On the other hand, JSON is much simpler, and the UNICORE commandline client already provides a very elaborate and complete JSON job description[12]. Thus, as a first step we implemented the JSON format, which is converted to JSDL and then submitted to the internal XNJS execution engine. As a trivial example,

```
{
  Executable: "/bin/ls",
  Arguments: ["-l"],
}
```

would be a valid job. Using curl as a simple HTTP client, the submission of a job in file "job.u" can be done simply using

```
curl -k -X POST -u user:pass
   https://localhost:8080/DEMO-SITE/services/rest/core/jobs/
  -H "Content-type: application/json"
 --data-binary @job.u
```

where username and password are conveniently given directly on the commandline. The server will reply with a "201 Created" status and the location of the new job:

```
HTTP/1.1 201 Created
Content-Length: 0
Date: Wed, 23 Jul 2014 13:19:56 GMT
Location: https://localhost:8080/DEMO-SITE/services/rest/core
          /jobs/b4cbb9eb-0560-4267-a205-8d25a9a2c89a
Server: Jetty(8.1.11.v20130520)
```

In UNICORE jobs are submitted to a TargetSystemService (TSS) instance. In the simple usage example outlined above, a TSS instance is selected automatically. In the future the full UNICORE feature set with TargetSystemFactory services and TargetSystemService instances will be supported.

As another simplification compared to the usual UNICORE clients, we will support sending file data inline with the job submission. This is intended for short data files such as scripts, large volumes of data will still need to be staged in. To avoid a denial of service by users sending huge data, a practical limit on this inline data will be imposed.

5 Summary and Outlook

We have extended UNICORE to allow building RESTful services that are fully consistent with the existing SOAP/WSRF based services. This includes the security stack used for RESTful services, which is fully compatible and consistent with the rest of the UNICORE world.

First applications include an authentication service for standalone use of UFTP, as well as initial implementations of the basic UNICORE services for storage and job management, including job submission.

Next steps will focus on finalizing the security architecture and implementing OpenID-Connect support, i. e. validating OIDC tokens and if required creating SAML trust delegation assertions from them using Unity.

Furthermore, the service APIs will be developed further, aiming at basic (but useful and productive) job submission and management capabilities for the first release with UNICORE 7.1 in Autumn 2014. The next full major release, UNICORE 8 is expected

in 2015, and will contain a complete implementation of the UNICORE service model in a RESTful way. Specifically, we want to support the creation of target system and storage instances in order to fully support cloud-like use cases.

An interesting and fruitful activity will be to compare the UNICORE approach to other RESTful service APIs like OCCI (compute) or CDMI (storage), and to learn how these have realised access to storage or handling of virtualisation.

The new RESTful APIs will open up the world of HPC and access to large-scale scientific data to a much wider audience, while keeping the UNICORE benefits of abstraction and ease of use. The RESTful APIs will allow to use simple authentication mechanisms like username password or OpenID-Connect, submit compute tasks to HPC machines, manage results, move data and much more, and will enable important new use cases.

Acknowledgements

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 604102 (Human Brain Project).

References

1. "Web Services Resource Framework." http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf. [accessed: 2014-07-22].
2. R. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
3. D. Crockford, "The application/json media type for javascript object notation (JSON)." RFC 4627, July 2006.
4. "OpenID Connect." http://openid.net/connect. [accessed: 2014-07-22].
5. "Human Brain Project." http://www.humanbrainproject.eu/. [accessed: 2014-07-22].
6. B. Schuller and T. Pohlmann, "UFTP: High-Performance Data Transfer for UNICORE," in *Proceedings of 7th UNICORE Summit 2011*, IAS Series, vol. 9, pp. 135–142, Forschungszentrum Jülich GmbH, 2011.
7. "Apache CXF web services framework." http://cxf.apache.org/. [accessed: 2014-07-22].
8. "Java API for RESTful Services (JAX-RS)." https://jax-rs-spec.java.net/. [accessed: 2014-07-22].
9. "Unity Identity Management Solution." http://www.unity-idm.eu/. [accessed: 2014-07-22].
10. "UNICORE Services Environment code repository." http://svn.code.sf.net/p/unicore/svn/wsrflite/trunk. [accessed: 2014-07-22].
11. "UFTP Authentication service code repository." http://svn.code.sf.net/p/unicore/svn/uftp/trunk/authserver. [accessed: 2014-07-22].
12. "UNICORE commandline client job description format." http://unicore.eu/documentation/manuals/unicore6/files/ucc/ucc-manual.html#ucc_jobdescription. [accessed: 2014-07-22].

Providing Integration of UNICORE Services in Private PaaS Platform

Gleb Radchenko[1] **and Dmitry Savchenko**[1]

System Programming Department,
South Ural State University, Chelyabinsk, Russia
E-mail: gleb.radchenko@susu.ru

The concept of grid computing has become a standard way of collaboration of scientific community computational resources while solving extra-large and resource-intensive tasks. On the other hand, the "cloud computing" concept is gaining increasing popularity in the field of provision of computing resources to an end-user on demand. PaaS (Platform as a Service) cloud model provides development, deployment, and administration tools to platform consumers, greatly simplifying the usage of remote computing resources. In this paper we propose the design and implementation of a Mjolnirr private cloud platform for development of the private PaaS cloud infrastructure. From a developer perspective, an application on the basis of the Mjolnirr platform is a set of independent components, which communicate through a message passing interface. Also, Mjolnirr platform allows integration of UNICORE services in cloud infrastructure, providing a mechanism to execute tasks in the UNICORE grid environment.

1 Introduction

In the last decade, the dominant method for providing remote computing resources to solve practical problems was the provision of such resources in accordance with the concept of cloud computing[2]. From the point of view of developers and professional users, the models of IaaS (Infrastructure as a Service) and PaaS (Platform as a Service) cloud solutions provide services that greatly simplify the usage of remote computing resources[1].

IaaS model is focused on providing consumers with low-level solutions, such as systems of processing, storage, and transmission of data, implemented in the framework of the concept of virtual machines. This approach assumes that each consumer of IaaS-resources will decide, what tools he will use to build and deploy his cloud application.

PaaS model provides a higher level of abstraction to the underlying computing resources, providing a transparent mechanism to deploy cloud applications using programming languages, libraries, services, and tools provided by the cloud platform. Thus, end users of PaaS solutions can significantly optimize the process of development, deployment, and execution of their applications in the cloud.

On the other hand, grid computing has become a standard way of collaboration of scientific community computational resources while solving extra-large and resource-intensive tasks. However, unlike electricity devices, integration of new resources to the grid-computing environment is not trivial.

To provide to the end user the simplicity of PaaS cloud solutions and computing power of existing grid systems, we propose a Mjolnirr cloud platform, providing the creation of private cloud PaaS-based systems, on the basis of component-oriented approach. Any library or Java application can be implemented on the basis Mjolnirr as a service. From a developer perspective, the Mjolnirr application is a set of independent services that communicate with each other via a message passing interface[3]. Also, Mjolnirr platform allows

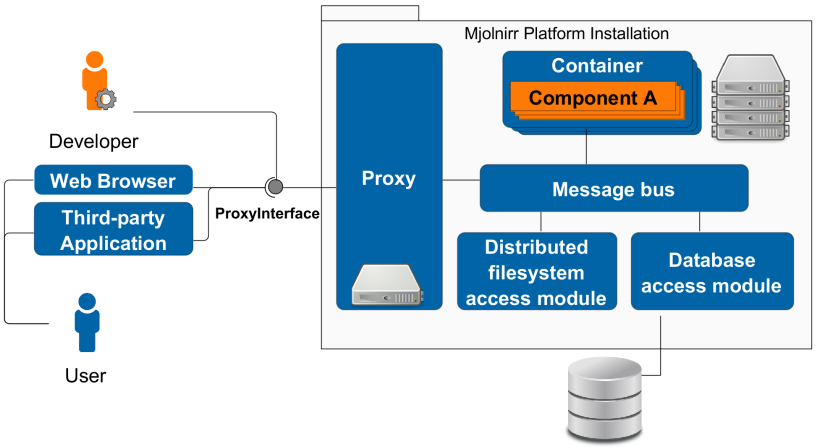

Figure 1. Mjolnirr platform architecture

integration of UNICORE services in cloud infrastructure, providing mechanism to execute tasks in the UNICORE grid environment.

The paper is organized as follows: the next section presents an overview of the Mjolnirr platform architecture and implementation. Section 3 describes the process of integration of UNICORE services in cloud infrastructure and an example of application that uses UNICORE services via the Mjolnirr interface. Finlay in Section 4 possible future extensions and enhancements are discussed.

2 Mjolnirr Platform Architecture

The Mjolnirr platform provides infrastructure for cloud applications development, including software developer kit, message brokering system, and browser support. Java-based application or library can be implemented as a Mjolnirr-based service.

The Mjolnirr platform includes the following components (Fig. 1):

- *Proxy* provides access to the cloud system for the external clients and manages the communication between cloud application components. It also hosts all of the system services (built-in modules for user authentication, distributed file system, database access etc.). Proxy is the only component that is accessible from the external network.
- *Container* is responsible for hosting of cloud applications components and message transmission. It can be deployed both on personal computers and on the computing server nodes.
- *Components* are custom applications, developed to run in Mjolnirr cloud environment. Each component has a unique name. UI components (applications) are multi-page applications.
- *Clients.* All client applications use encrypted channel to communicate with the proxy. Each client should use a certificate for authentication.

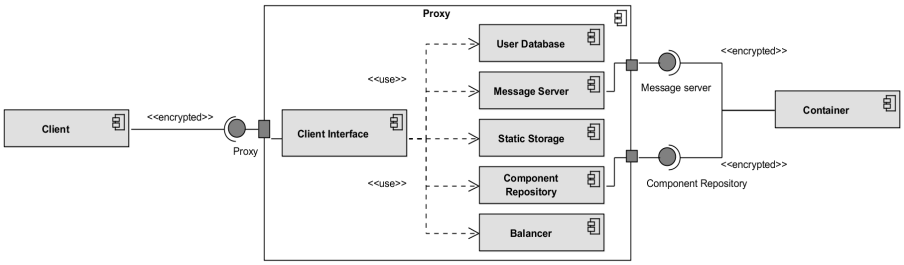

Figure 2. Proxy architecture

2.1 Proxy

The Proxy component (Fig. 2) provides access to Mjolnirr system from the external network. The Proxy performs the following actions:

- it stores and provides static resources (page layout descriptions, images etc.) of the deployed cloud applications in the Static Storage;
- it acts as a messaging server for the components of cloud applications;
- it handles client's requests to the Client Interface;
- it performs the authorization and authentication of users.

The external Proxy interface provides the following methods as RESTful API:

- *getUI (applicationName, pageName)* – returns the page layout description;
- *getResourceFile (applicationName, resourceName)* – gets the static file;
- *processRequest (componentName, methodName, args)* processes a client request, redirecting it to the first free suitable component. This method can be called directly from the application page (e. g. with JavaScript).

2.2 Container

The container (Fig. 3) provides cloud application component hosting. The container provides an API for remote components instances method invocation. The Mjolnirr installation can have any number of containers.

Any Mjolnirr-based application consists of independent components, which use a built-in messaging system, implemented in the basis of the Publisher-Subscriber pattern. The Proxy is responsible for message queue maintenance. The Message Server of the Proxy provides publisher-subscriber messaging service for cloud application components. Mjolnirr Containers subscribe to Message Channels that operate as a broadcast delivery. Any message sent to the Message Channel will be transmitted to the subscribers of this channel.

Each cloud application instance is subscribed on two types of Message Channels:

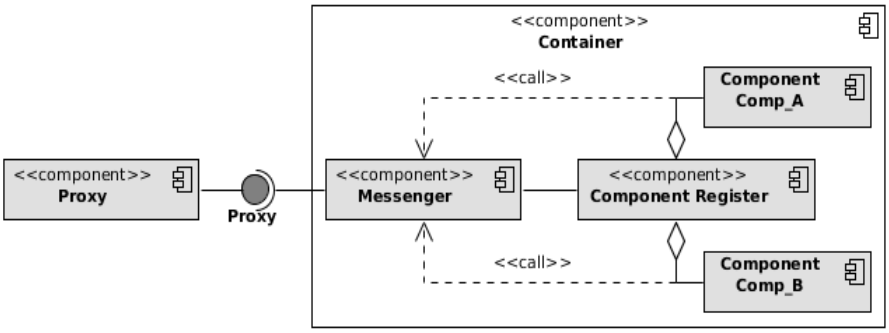

Figure 3. Container architecture

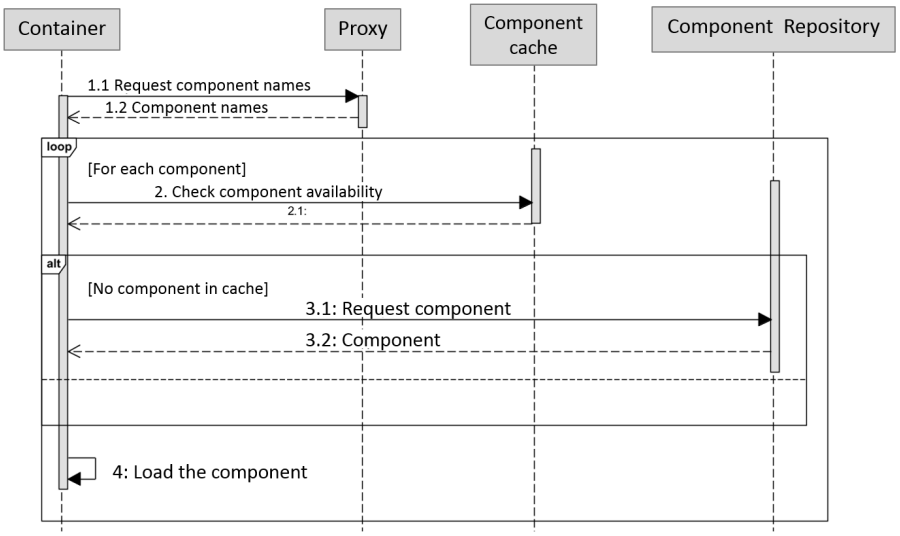

Figure 4. Container initialization process

- *Component Public Channel:* every instance of the cloud application component is subscribed on this public channel. This a listener channel – when any message comes to this channel, the appropriate component instance will be invoked.
- *Instance Private Channel:* provides direct communication between instances.

When container starts, it performs several initialization steps. The order of the container initialization (Fig. 4):

- Container registers in the proxy database and receives the list of components to load in response;
- For each component from the list:

– The container checks its local cache, for each missing package container downloads it from the proxy;
– Container runs the component;
– Container subscribes on the Component Public Chanel and Instance Private Channel for the loaded component.

In addition, container has an opportunity to work in stand-alone mode. In this mode, the container does not support communication with other containers and acts as a stand-alone computing system (container and proxy at the same time).

2.3 Components

From the developer's point of view, Mjolnirr cloud application is a collection of independent components communicating by message exchange. Components are represented as a package that contains the following information:

- *manifest*, that provides the interface of the component, including description of provided methods and their parameters;
- *executables* to handle incoming requests;
- *static files*, used in pages rendering (images, page layout descriptions, and scripts) for UI provision.

Each component can be:

- *Application Component*: provides the user interface definition, scripts, styles, and UI-specific actions. Optionally contains complex logic.
- *Module Component*: represents a single entity in the domain logic of the application. The Module Component provides data processing and storage, but does not provide interface and static files.

To implement Mjolnirr components API, we developed an interface definition language based on Java 6 annotation mechanism. Figure 5 shows an example of the interface for a Calculator component class.

A *@MjolnirrComponent* annotation denotes the interface class of component and contains the following fields:

- *ComponentName* - the name of the component in the system. Under this name the feature will be available for remote calls.
- *InstancesMinCount* - the minimum number of instances of a component in the system.
- *InstancesMaxCount* - the maximum number of instances of a component in the system.
- *MemoryVolume* - the amount of memory in megabytes required by the component for proper operation.

```
MjolnirrComponent(
    componentName = "calculator",
    instancesMinCount = 1,
    instancesMaxCount = 255,
    memoryVolume = 128)
public class Calculator
    extends AbstractApplication {
    private ComponentContext context;
    @MjolnirrMethod
    public String calculate(
            String expression)
        return Helper.calculate(expression);
    }

    @Override
    public void initialize(
        ComponentContext context) {
        this.context = context;
    }
}
```

Figure 5. Component interface definition

A *@MjolnirrMethod* annotation defines a separate method of the interface class as exportable. This annotation has one optional field: *executionTime*. The *executionTime* value indicates the maximum length of a method of execution of this method in seconds. The default value is 30 seconds.

Container class parses annotation interface for each downloadable component, reading the description of the methods provided by the component.

3 Mjolnirr UNICORE Integration Module

The Mjolnirr platform provides a UNICORE 7.0.2 integration module. UNICORE integration module is an interface for UNICORE installation (see Fig. 6).

UNICORE integration module has its own UNICORE certificate and works as an external client. The module does not use UNICORE authentication system, because Mjolnirr itself is not based on UNICORE, so the integration module is just a plug-in to execute tasks in the grid environment.

UNICORE component is an entry point to the whole UNICORE installation, in other words, it is the UNICORE Gateway. Integration module is based on UCC sources and uses standard UNICORE protocols to communicate with the grid environment. UNICORE integration module retrieves the list of all available sites and scans each site to find the proper application. When the application is found, UNICORE integration module uses the URL of the site, which hosts the target application (Fig. 7).

Mjolnirr UNICORE integration module provides the interface, shown in Fig. 8. As stated before, UNICORE integration module can be called in a standard Mjolnirr way. To send the task to the UNICORE installation, client application must submit the following attributes:

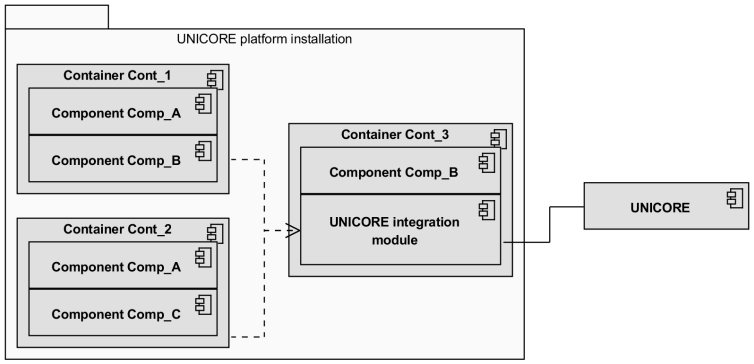

Figure 6. UNICORE integration module

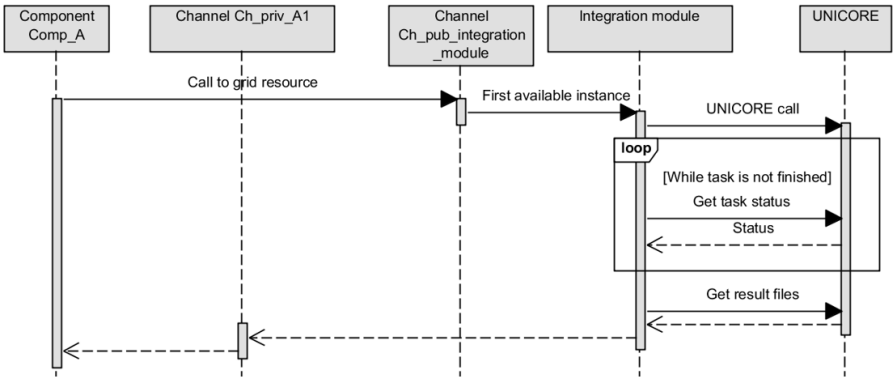

Figure 7. UNICORE Module call

- *appName* - a name of the UNICORE application to run;
- *appVersion* - a version of the application to run;
- *parameters* - a map of the parameters to pass to UNICORE;
- *inputFiles* - a list of files to send into the UNICORE installation (those files will be added to inputs automatically);
- *inputs* - a list of files to be processed by the UNICORE-hosted application;
- *outputs* - a list of files to be downloaded from UNICORE after application execution.

Those methods can be called in a standard Mjolnirr way, via built-in messaging system.

```java
@MjolnirrComponent(componentName = "unicore", instancesMaxCount = 1)
public class UnicoreModule extends AbstractModule {

    /* Execute UNICORE application
     * @param appName - Name of the application hosted in UNICORE
     * @param appVersion - Version of the application hosted in UNICORE
     * @param parameters - Params to pass to the application.
     *        All input files must be marked as 'fileX', X - number
     * @param inputFiles - List of input files for the application
     * @param inputs - List of files names to send to UNICORE.
     *        All the inputFiles will be uploaded by default
     * @param outputs - List of files to download from UNICORE.
     *
     * @returns List of files, downloaded from UNICORE
     */
    @MjolnirrMethod(maximumExecutionTime = 9999)
    public List<File> run(String appName,
                String appVersion,
                Map<String, String> parameters,
                List<File> inputFiles,
                List<String> inputs,
                List<String> outputs) throws Exception;

    @Override
    public void initialize(ComponentContext componentContext;
}
```

Figure 8. UNICORE Module interface

3.1 UNICORE Integration Test

To test the UNICORE integration module, we implemented an OpenMP test stand. Using this application, user can upload his own OpenMP program source to the platform and then get the detailed output for execution on N thread for N=1..16. An output example is shown in Fig. 9.

In this OpenMP test stand, the source file, which was uploaded by the user, is transmitted to the UNICORE application. This application does the following steps:

- Source file is compiled using GCC compiler with required flags (-fopenmp);
- Application sets the environment variables to specify current thread count;
- Application is run;
- Execution log is returned to Mjolnirr.

4 Conclusions and Future Work

In this work, we presented a Mjolnirr cloud PaaS platform, allowing development of scalable distributed Java-based applications on the basis of Publisher-Subscriber pattern message exchange. We developed a UNICORE integration module that allows using UNICORE services as components of such cloud applications. We presented the architecture, mechanisms and example of test Mjolnirr application that use the UNICORE/X sites as cloudlets.

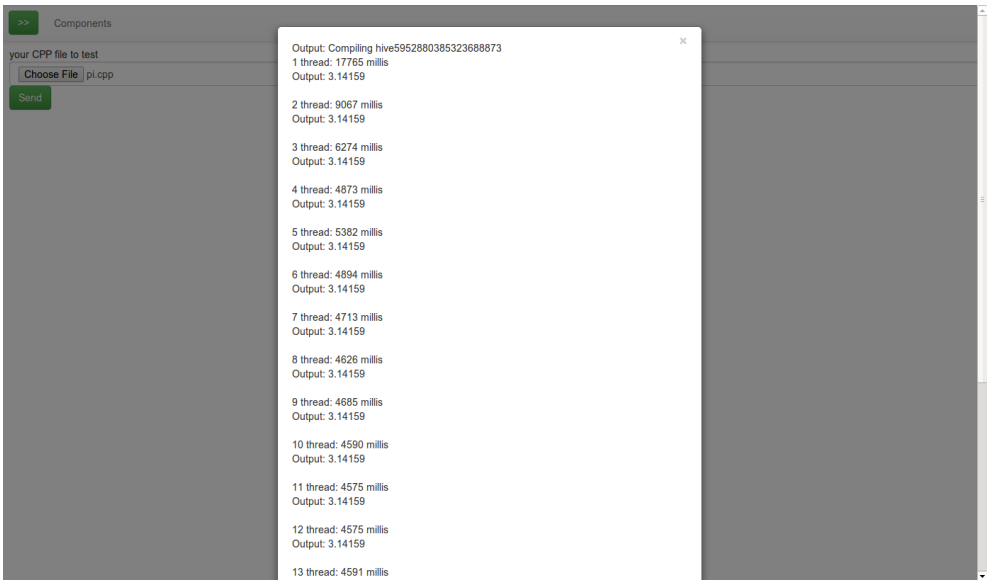

Figure 9. UNICORE-based OpenMP test stand

As a further development of the Mjolnirr platform we will investigate and implement application-level migration support, integration with the advanced resource monitoring systems, flexible adaptation to load changes, advanced system security and application store. The application store will reduce the number of duplicate software products and simplify the creation of individual business infrastructure to meet the needs of a particular company.

Acknowledgements

The reported study was partially supported by RFBR, research project No. 14-07-00420-a and by Grant Fund for Assistance to Small Innovative Enterprises in Science and Technology research project No. 0000829

References

1. M. Hogan, A. Sokol, *NIST cloud computing standards roadmap - Version 2*, NIST Special Publication, 500-291, National Institute of Standards & Technology, 113 p, 2013.
2. P. Mell, T. Grance, *The NIST definition of cloud computing*, NIST special publication, 800-145, National Institute of Standards & Technology, 7 p, 2011.
3. D. Savchenko, G. Radchenko, *Mjolnirr: A Hybrid Approach to Distributed Computing Architecture and Implementation*, in: CLOSER 2014 Proc. of the 4th International Conference on Cloud Computing and Services Science. pp. 445-450, 2014.

Resource Scheduling Algorithm in Distributed Problem-Oriented Environments

Anastasia Shamakina and Leonid Sokolinsky

South Ural State University,
76, Lenin av., 454080 Chelyabinsk, Russian Federation
E-mail: {shamakinaav, Leonid.Sokolinsky}@susu.ru

Nowadays a large number of scheduling algorithms for the use in distributed computing environments. Only a small part of these algorithms takes into account the problem-oriented specificity. We have created a mathematical model of a computing application, which represents a workflow. The application is modelled as a weighted labelled directed acyclic graph of a job. Each task (that is a graph vertex) is labelled by a pair of natural numbers, the first one of which sets the executing time of the task on a single processor core, and the second one sets the maximum number of cores on which the task demonstrates a nearly-linear speedup. The edge weights define the amount of data, transferred between the tasks. A new Problem-Oriented Scheduling (POS) algorithm for distributed problem-oriented environments is proposed. The POS algorithm is scheduling algorithm enables to schedule tasks to run on several processor cores with task scalability limitations. The POS algorithm is designed to create a system of intelligent preparation and scheduling a workflow in order to increase the efficiency of the supercomputer complexes with cluster architecture.

1 Introduction

Today, there is a large number of scheduling algorithms [1-7] which are targeted for the use in distributed computing environments. Only a small part of these algorithms takes into account the problem-oriented specificity of workflows in complex applications. This specificity is expressed in the way of setting the execution time of a task and the amount of transmitted data.

One of the most significant achievements in this area is a Dominant Sequence Clustering (DSC) algorithm [5] invented by Yang T. and Gerasoulis A. The DSC scheduling algorithm introduces a job as a directed acyclic graph whose nodes are tasks and whose edges represent a data flow. While job scheduling, the DSC algorithm takes into account the execution time of tasks and the amount of transmitted data. A significant limitation of this algorithm applied in modern cluster computing systems is that each task can be executed only on one processor core.

We have developed a new Problem-Oriented Scheduling (POS) resource scheduling algorithm for distributed computing environments, which, unlike the DSC algorithm, allows to schedule a task to run on several processor cores with the limitations on scalability of this task.

2 Mathematical Job Model

Before a description of the POS scheduling algorithm we present a mathematical model of a job for which we need the following basic definitions.

2.1 Basic Definitions

A directed graph is called quadruple $G = \langle V, E, init, fin \rangle$, where V is a vertices set; E is an edges set; $init : E \to V$ is a function which determines *an initial vertex* of an edge; $fin : E \to V$ is a function which determines *a final vertex* of an edge.

The vertices $\nu, \nu^{'} \in V$ are called *adjacent*, if

$$\exists e \in ((\nu = init(e) \& \nu^{'} = fin(e)) \bigvee (\nu = fin(e) \& \nu^{'} = init(e))), \tag{1}$$

in other words, there is an edge e connecting these vertices. If $\nu = init(e) \& \nu^{'} = fin(e)$, we denote it as follows: $(\nu, \nu^{'}) = e \in E$ and we say that the vertices $\nu, \nu^{'}$ are *incident* to the edge e.

Let us take the vertices $\nu, \nu^{'} \in V$ and the number $n \geq 1$. An ordered sequence of edges $(e_1, e_2, \ldots, e_n) \in E^n$ is called *a path of a length n* from the vertex ν to the vertex $\nu^{'}$, if $\nu = init(e_1)$, $\nu^{'} = fin(e_n)$ and $fin\,(e_i) = init(e_{i+1})$ for all $i \in \{1, \ldots, n-1\}$. If $(e_1, e_2, \ldots, e_n) \in E^n$ is a path from the vertex ν to the vertex $\nu^{'}$, then *a return path* from the vertex $\nu^{'}$ to the vertex ν is called an ordered sequence of edges $(e_n, e_{n-1}, \ldots, e_1) \in E^n$. The path $(e_1, e_2, \ldots, e_n)$ is called *simple one*, if all vertices $init(e_1), init(e_2), \ldots, init(e_n)$ different from each other and if all vertices $fin(e_1), fin(e_2), \ldots, fin(e_n)$ also distinct. *A circle* is called a simple path from some vertex to itself. A directed graph is called *acyclic one*, if it does not contain cycles [8].

The vertices $\nu, \nu^{'} \in V$ are called *independent ones*, if there is no simple or return paths from the vertex ν to the vertex $\nu^{'}$. Otherwise, the vertices $\nu, \nu^{'}$ are *dependent*.

Let us take a directed graph $G = \langle V, E, init, fin \rangle$. *The weighting of the graph* G is called a function $\delta : E \to \mathbb{Z}_{\geq 0}$. *A layout of the graph* G is called a function

$$\gamma : V \to \mathbb{N}^2. \tag{2}$$

2.2 Computing Environment Model

Now, we are ready to give a mathematical definition of a job graph in a distributed computing environment.

A job graph is called a marked-up weighted directed acyclic graph,

$$G = \langle V, E, init, fin, \delta, \gamma \rangle,$$

where V is a set of vertices which correspond to tasks, E is a set of edges which correspond to data flows. *The weighting function* $\delta\,(e)$ of the edge e determines the amount of transmitted data from a task associated with the vertex $init(e)$ to a task associated with the vertex $fin(e)$. *A layout*

$$\gamma\,(\nu) = (m_\nu, t_\nu) \tag{3}$$

determines the maximum number of processor cores m_ν on which the task ν has *a nearly-linear speedup* of the execution time t_ν of task ν on a single core. In this model, we assume that *the computational cost* $\chi\,(\nu, j_\nu)$ of the task ν on $j_{\nu-th}$ processor cores is determined by the following formula:

$$\chi(\nu, j_\nu) = \begin{cases} t_\nu / j_\nu, & \text{if } 1 \leq j_\nu \leq m_\nu; \\ t_\nu / m_\nu, & \text{if } m_\nu < j_\nu. \end{cases} \tag{4}$$

In other words, the execution time decreases directly proportional to the increasing the number of processor cores in the range from 1 to m_ν. We will not get the acceleration by increasing the number of processor cores in the range from m_ν to $+\infty$.

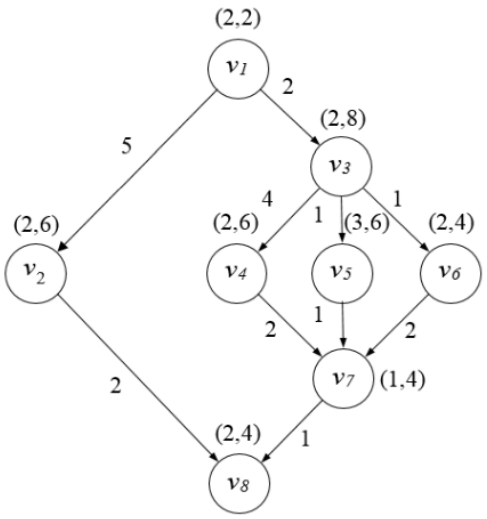

Figure 1. An example of a job graph.

Fig. 1 shows an example of job graph consisting of 8 vertices. A label is a pair of numbers (m_ν, t_ν) specified for each of the vertices. Each edge has a weight. The weight $\delta(e)$ determines the amount of data transmitted to the edge e.

The compute node P is an ordered set of processor cores $\{c_0, \ldots, c_{d-1}\}$.

The computer system $\mathfrak{P}$ is an ordered set of compute nodes $\{P_0, \ldots, P_{k-1}\}$. In reality, the computer system may be a distributed computing system, combining several computing clusters, each of which is a separate node on this system.

The clustering is called single-valued transformation $\omega : V \to \mathfrak{P}$ of vertices set V of the job graph G on a set of computational nodes $\mathfrak{P}$.

The computer system $\mathfrak{P} = \{P_0, \ldots, P_{k-1}\}$, consisting of k computational nodes, is given. *The cluster* W_i is a set of all vertices that are displayed on the computational node $P_i \in \mathfrak{P}$:

$$W_i = \{\nu \in V | \omega(\nu) = P_i \in \mathfrak{P}\}. \tag{5}$$

We have the following properties:

$$W_i \bigcap W_j = \emptyset \text{ for } i \neq j; \tag{6}$$

$$V = \bigcup_{i=0}^{k-1} W_i. \tag{7}$$

Let us consider the job graph $G = \langle V, E, init, fin, \delta, \gamma \rangle$ with the clustering function ω. We will call such job graph as *clustered one* and denote as $G = \langle V, E, init, fin, \delta, \gamma, \omega \rangle$.

In this model, we assume that the transfer time of any amount of data between nodes belonging to the same cluster is close to zero, and the transmission of data between nodes belonging to different clusters is proportional to the volume of transmitted data with coefficient 1. Based on this, we can define *the communication cost function* $\sigma : E \to \mathbb{Z}_{\geq 0}$, which computes the communication cost (time) of transmitted data along the edge $e \in E$ as follows:

$$\sigma(e) = \begin{cases} 0, & \text{if } \omega(init(e)) = \omega(fin(e)); \\ \delta(e), & \text{if } \omega(init(e)) \neq \omega(fin(e)). \end{cases} \tag{8}$$

The clustered graph $G = \langle V, E, init, fin, \delta, \gamma, \omega \rangle$ is given. *A schedule* is called single-valued transformation, $\xi : V \to \mathbb{Z}_{\geq 0} \times \mathbb{N}$, which maps the casual vertex $v \in V$ on a pair of numbers,

$$\xi(v) = (\tau_v, j_v), \tag{9}$$

where τ_v determines the launch time of the task v and j_v is a number of processor cores allocated to task v. We denote by s_v the stop time of the task v. Then

$$s_v = \tau + \chi(v, j_v), \tag{10}$$

where χ is the computational cost function defined by the formula 4. A schedule is called *valid one*, if it satisfies the following conditions:

$$\forall\, e \in E \ \left(\tau_{fin(e)} \geq \tau_{init(e)} + \chi\left(init(e), j_{init(e)}\right) + \sigma(e)\right); \tag{11}$$

$$\forall\, v \in V \left(j_v \leq m_v\right); \tag{12}$$

$$\forall\, t \in \mathbb{N} \left(\forall\, i \in [0, \ldots, k-1] \left(\sum_{v \in W_i \& \tau_v < t \leq s_v} j_v \leq |P_i| \right) \right). \tag{13}$$

The condition 11 means that for any two adjacent vertices $v_1 = init(e)$ and $v_2 = fin(e)$ the start time v_2 cannot be less than the sum of the following values: the start time of the task v_1, the execution time of the task v_1, and the communication cost of the edge e. The condition 12 means that the number of cores allocated the task v_1, does not exceed the boundaries of linear scalability, which sets the layout γ in the context of the formula 3. The condition 13 means that at any time t the total number of processor cores allocated by tasks on the node number i cannot exceed the number of cores on the same node. In what follows we consider only valid schedules, unless explicitly stated otherwise.

The clustered graph $G = \langle V, E, init, fin, \delta, \gamma, \omega \rangle$ with the specified schedule ξ is called *a scheduled graph*, $G = \langle V, E, init, fin, \delta, \gamma, \omega, \xi \rangle$.

A tiered-and-parallel form of a graph [9] is called a partition of vertices set V of the directed acyclic graph $G = \langle V, E, init, fin \rangle$ into renumbered subsets (tiers) L_i, where $i = 1, \ldots, r$, satisfying the following properties:

$$\left.\begin{array}{l} V = \bigcup\limits_{i=1}^{r} L_i; \\ \forall i \neq j \in \{1, \ldots, r\} (L_i \bigcap L_j = \emptyset); \\ \forall (v_1, v_2) \in E (\forall i \neq j \in \{1, \ldots, r\} (v_1 \in L_i \& v_2 \in L_j \Rightarrow i < j)). \end{array}\right\} \tag{14}$$

The last condition means that if there is an edge from the vertex v_1 to the vertex v_2, then the vertex v_2 must be placed on a tier with a large number in relation to a tier on which the vertex v_1 is placed. A number of vertices in a tier L_i is called its *width*. A number of tiers in a tiered-and-parallel form is called its *height*, and the maximum of a width of its tiers is called *a width of a tiered-and-parallel form*. A tiered-and-parallel form is called *canonical one* [10], if all input vertices (without input edges) belong to the tier 1 and the maximum path length, ending at a vertex owned tier k, equals $k-1$.

Fig. 2 presents an example of a canonical tiered-and-parallel form consisting of 5 tiers for the graph shown in Fig. 1.

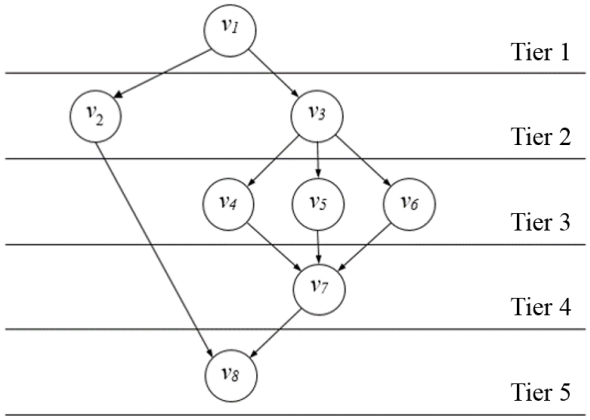

Figure 2. An example of a canonical tiered-and-parallel form.

Let us take a simple path, $y = (e_1, e_2, \ldots, e_n)$, in the scheduled graph $G = \langle V, E, init, fin, \delta, \gamma, \omega, \xi \rangle$. Since the graph G is acyclic, any path in it, including the path y, will be the simple one. *The path cost* $u(y)$ has the following value:

$$u(y)=\chi\left(fin(e_n), j_{fin(e_n)}\right) + \sum_{i=1}^{n}\left(\chi\left(init(e_i), j_{init(e_i)}\right) + \max(\sigma\left(e_i\right), \tau_{fin(e_i)} - s_{init(e_i)})\right), \quad (15)$$

where χ is a function defined by the formula 4, whose value is a computational cost of a vertex; σ is a function defined by the formula 8, whose value is a communication cost of edge; j_v and τ_v are defined by the formula 9; s_v is defined by the formula 10.

A set Y of all simple paths in scheduled graph $G = \langle V, E, init, fin, \delta, \gamma, \omega, \xi \rangle$ is given. The simple path $\bar{y} \in \mathrm{Y}$ is called *a critical path*, if it has a maximum value:

$$u(\bar{y})=\max_{y \in Y} u(y)\ . \quad (16)$$

3 Problem-Oriented Scheduling Algorithm

In this section, we will describe a new POS (Problem-Oriented Scheduling) algorithm for the scheduling in distributed problem-oriented environments. A distinctive feature of the POS algorithm is that it takes into account the knowledge about the specificity of a subject area. In the model described in Subsection 2.2 this specificity is expressed in the way of setting the execution time of a task and the amount of transmitted data.

To simplify a description and an understanding of the POS algorithm, we will use the three-level structure of procedures representing the POS algorithm. The procedure of the first level is *the main procedure*. Some steps in the first level will be described in the second level. We will highlight such steps in bold. A similar approach will be used in a description of the second level.

3.1 Main Procedure

Let us consider the computer system $\mathfrak{P} = \{P_0, \ldots, P_{k-1}\}$ in the form of an ordered set of computing nodes. The job graph $G = \langle V, E, init, fin, \delta, \gamma \rangle$ is given. We assume that the following conditions are fulfilled:

$$|V| \leq |\mathfrak{P}|, \tag{17}$$

$$\forall v \in V(\forall P \in \mathfrak{P}(m_v \leq |P|)), \tag{18}$$

where m_v is a boundary linear scalability defined by the layout function γ. We define for graph G a canonical tiered-and-parallel form with tiers $L_i (i = 1, \ldots, r)$. Then we enumerate vertices $V = (v_1, \ldots, v_q)$ of the graph G so that the following property is fulfilled:

$$\forall i, j \in \{1, \ldots, q\}((v_i \in L_a \& v_j \in L_b \& a < b) \Rightarrow i < j), \tag{19}$$

this has to result that vertices with large numbers are arranged on the lower tiers.

Tab. 1 presents the main procedure in the most general form.

Step 1. **Initial configuration** $G_0 = \langle V, E, init, fin, \delta, \gamma, \omega_0, \xi_0 \rangle$;
Step 2. $i := 0$;
Step 3. **Iterate over configurations: from** $G_i = \langle V, E, init, fin, \delta, \gamma, \omega_i, \xi_i \rangle$ **to** $G_{i+1} = \langle V, E, init, fin, \delta, \gamma, \omega_{i+1}, \xi_{i+1} \rangle$**.** **The considered edges will be marked as examined;**
Step 4. If there are still unconsidered edges remaining, then
Step 4.1. $i := i + 1$;
Step 4.2. go to step 3;
Step 5. **Compressing** $G_{i+1} = \langle V, E, init, fin, \delta, \gamma, \omega_{i+1}, \xi_{i+1} \rangle$**;**
Step 6. The end of the procedure.

Table 1. The main procedure.

Thus, the purpose of the main procedure is to construct a sequence of configurations. In this case, during the transition from one to another configuration at least one edge of

a graph is marked as examined. Since the number of edges is finitely, the procedure is completed at some iteration. Last built configuration G_{i+1} is selected as the resultant.

3.2 Initial Configuration

Tab. 2 presents the procedure of preparation of the initial configuration G_0.

Step 1.1. We define a function of an initial clustering as follows:
$\forall i \in \{1, \ldots, q\}(\omega_0(v_i) = P_{i-1})$.
that is, each vertex is displayed on a separate computing node,
and accordingly, each cluster includes only one vertex.

Step 1.2. We specify τ_v and j_v for an initial schedule $\xi_0(v) = (\tau_v, j_v)$.
Then recursively define start time τ_v by tiers of a tiered-and-parallel form:

$$\left.\begin{array}{l} \forall v \in L_1(\tau_v := 0); \\ \forall v \in L_{i>1}(\tau_v := \max\limits_{v'' \in L_i; v' \in L_{j<i}} (\lambda(v', v''))). \end{array}\right\}$$

Here

$$\lambda(v', v'') = \begin{cases} s_{v'}, if\ (v', v'') \notin E; \\ s_{v'} + \sigma((v', v'')), if\ (v', v'') \in E. \end{cases}$$

where $s_{v'}$ is defined by the formula 10. We determine the number of processor cores j_v allocated to the vertex v as follows: $\forall v \in V(j_v = m_v)$.

Step 1.3. $G_0 = \langle V, E, init, fin, \delta, \gamma, \omega_0, \xi_0 \rangle$.

Step 1.4. The end of the procedure.

Table 2. The procedure for preparing the initial configuration G_0.

3.3 G_{i+1} Configuration

We define *a subcritical path* as a path having the maximum cost and containing at least one unexamined edge. Tab. 3 presents the procedure for preparing the configuration G_{i+1}.

3.4 ξ_{i+1} Schedule

We introduce the following notation: $T(x)$ is the tier number, which the vertex x owns; $W_{\omega_i(x)} = \{v | v \in V, \omega_i(v) = \omega_i(x)\}$ is the cluster, which the vertex x owns (Fig. 4).

3.5 G_{i+1} Compressing

The aim of the compressing procedure is to minimize the number of involved computing nodes. This procedure applies only to clusters containing one vertex. This limitation is motivated by the fact that if there are two adjacent vertices in a cluster, then the movement one of them to another cluster may increase the total task time.

Let us give some explanation to the compressing procedure. Step 5.2 organizes a loop, building cluster set $\mathfrak{M}$, which is a partition of the tasks set V (Fig. 5).

Step 3.1. Find a subcritical path for the graph $\tilde{y}_i = (e_1, \ldots, e_n)$;
Step 3.2. Find the first unexamined edge e_j $(1 \le j \le n)$ from the beginning of path $\tilde{y}_i$ and marking it as examined one;
Step 3.3. If $i = 0$, then mark vertex $init(e_j)$ as fixed one;
Step 3.4. If vertices $init(e_j)$ and $fin(e_j)$ are fixed, then go to step 3.14;
Step 3.5. If the vertex $fin(e_j)$ is not fixed, then $v'' := fin(e_j), v' := init(e_j)$;
Step 3.6. If the vertex $init(e_j)$ is not fixed, then $v'' := init(e_j), v' := fin(e_j)$;
Step 3.7. Specify the function of clustering ω_{i+1}, which equals the function of clustering ω_i, excepting $\omega_{i+1}(v'') := \omega_i(v')$;
Step 3.8. **Prepare the schedule** ξ_{i+1};
Step 3.9. $G_{i+1} = \langle V, E, init, fin, \delta, \gamma, \omega_{i+1}, \xi_{i+1} \rangle$;
Step 3.10. Find the critical path $\bar{y}_i$ in the graph G_i;
Step 3.11. Find the critical path $\overline{y_{i+1}}$ in the graph G_{i+1};
Step 3.12. If $u(\overline{y_{i+1}}) \le u(\bar{y}_i)$, then go to step 3.16;
Step 3.13. $G_{i+1} := G_i$;
Step 3.14. If there are unexamined edges in path $\tilde{y}_i$ then go to step 3.2;
Step 3.15. If there are unexamined edges in graph G_i, then go to step 3.1;
Step 3.16. The end of the procedure.

Table 3. The procedure for preparing the configuration G_{i+1}.

Step 3.8.1. $R := W_{\omega_i(v')} \bigcap L_{T(v'')}$;
Step 3.8.2. If $R = \emptyset$ or $\sum\limits_{v \in R} j_v \le |P_{\omega_i(v')}|$, then go to step 3.8.7;
Step 3.8.3. For $h = q, \ldots, T(fin(e_j)) + 1$ perform $L_{h+1} := L_h$;
Step 3.8.4. $L_{T(v'')+1} := \{v''\}$;
Step 3.8.5. $L_{T(v'')} := L_{T(v'')} \setminus \{v''\}$;
Step 3.8.6. $q := q + 1$;
Step 3.8.7. Specify new schedule ξ_{i+1} by calculating the start time τ_v of all vertices $v \in V$ using the formula (21);
Step 3.8.8. Mark vertex v'' as fixed one;
Step 3.8.9. The end of the procedure.

Table 4. The procedure for preparing the schedule ξ_{i+1}.

Step 5.1. $\mathfrak{M} := \emptyset$;
Step 5.2. For each vertex $v' \in V$ perform a loop
Step 5.2.1. $W = \{v | v \in V, \omega_{i+1}(v) = \omega_{i+1}(v')\}$;
Step 5.2.2. If $W \in \mathfrak{M}$, then go to next iteration of the loop;
Step 5.2.3. $\mathfrak{M} := \mathfrak{M} \bigcup \{W\}$;
Step 5.3. The end of the loop (step 5.2);

Table 5. Building cluster set $\mathfrak{M}$.

Tab. 6 presents the compressing procedure of graph G_{i+1}. Step 5.4 computes set $\mathfrak{E}$ of *single clusters* (clusters containing only one vertex) and set $\mathfrak{V}$ of *multiclusters* (clusters comprising two or more vertices). Step 5.5 loops through all single clusters. For each single cluster we find a tier of a tiered-and-parallel form, to which this single cluster belongs. Within this tier we try to combine the single cluster with some multicluster. It is possible, if a multicluster does not use all processor cores and the number of free cores is sufficient to perform the single cluster.

Step 5.4. $\mathfrak{E} := \{W \in \mathfrak{M} || W_a| = 1\}$; $\mathfrak{V} := \{W \in \mathfrak{M} || W_a| > 1\}$;
Step 5.5. For each $W' \in \mathfrak{E}$ perform a loop
Step 5.5.1. For $l = 1, \ldots, r$ perform a loop
Step 5.5.1.1. If $W' \bigcap L_l = \emptyset$, then go to next iteration of the loop;
Step 5.5.1.2. For each $W'' \in \mathfrak{V}$ perform a loop
Step 5.5.1.2.1. If $W'' \bigcap L_l = \emptyset$, then go to next iteration of the loop;
Step 5.5.1.2.2. Get $v' \in W'$ and $v'' \in W''$;
Step 5.5.1.2.3. If $j_{v'} + \sum\limits_{v \in W'' \bigcap L_l} j_v > \omega_{i+1}(v'')$, then transfer to next iteration of the loop;
Step 5.5.1.2.4. $i := i + 1$;
Step 5.5.1.2.5. Specify the function of clustering ω_{i+1}, which equals the clustering function ω_i, exepting $\omega_{i+1}(v') := \omega_i(v'')$;
Step 5.5.1.2.6. $W'' := W'' \bigcup W'$;
Step 5.5.1.2.7. Go to step 5.5.3;
Step 5.5.1.3. The end of the loop (step 5.5.1.2);
Step 5.5.2. The end of the loop (step 5.5.1);
Step 5.5.3. Go to next iteration of the loop;
Step 5.6. The end of the loop (step 5.5);
Step 5.7. The end of the procedure.

Table 6. The procedure of compressing G_{i+1}.

4 Integration with UNICORE

The POS algorithm has been implemented as a Grid service of the Distributed Virtual Test Bed (DiVTB) system [12, 11] on the basis of the UNICORE platform [13, 14] using Java programming language.

The DiVTB (Distributed Virtual Test Bed) is a software system that ensures the development and operation of distributed virtual test beds. DiVTB provides for a task-oriented approach of solving specific classes of problems in engineering design through resources supplied by grid computing environments.

The DiVTB system includes the following components (Fig. 3).

A DiVTB Developer which is a web-application that provides for the development for DiVTBs for a grid environment. An application programmer is given an opportunity to visually design the workflow for simulation, to create a file template of a task statement, to generate a set of parameters of a simulated task. The system also enables a function of the export DiVTB to a DiVTB Server where they can be launched for task calculation.

A DiVTB Portal which is a web-application for an engineer, which provides for start-up and reception of simulation results of virtual experiments. The DiVTB Portal ensures of authentication, user account management and a user interface for virtual experiments by means of DiVTB. The DiVTB Portal incorporates a built-in web-form generator which automatically generates a user interface on the basis of a relevant DiVTB's parameter description.

A DiVTB Viewer which is an auxiliary service for interactive visualisation of distributed virtual test beds. The DiVTB Viewer is designed to check for a correct display of generated problem-oriented shells of a DiVTB.

A DiVTB Server which is an environment for storing of DiVTBs and managing virtual experiments. The DiVTB Server ensures the execution of a DiVTB workflow, including the formatting of a file of task statement for each individual computing service in a grid environment on the basis of experimental parameters defined by an engineer, as well as the obtaining of simulation results and transfer of results to an engineer.

A DiVTB Broker which implements automated registration, search for, and allocation of grid resources to perform the actions of engineering simulation. The DiVTB Broker maintains a permanent record of grid resources which are available in the computing environment, and provides sets of computing resources at the demand of the DiVTB Server for carrying out computational experiments [6]. The DiVTB Broker component obtains information about existing applications of a grid computing environment and resources from the Registry and UNICORE/X Site components of the UNICORE.

In the Fig. 3 the components of the UNICORE platform that intend to cooperate with the DiVTB system are marked with red colour.

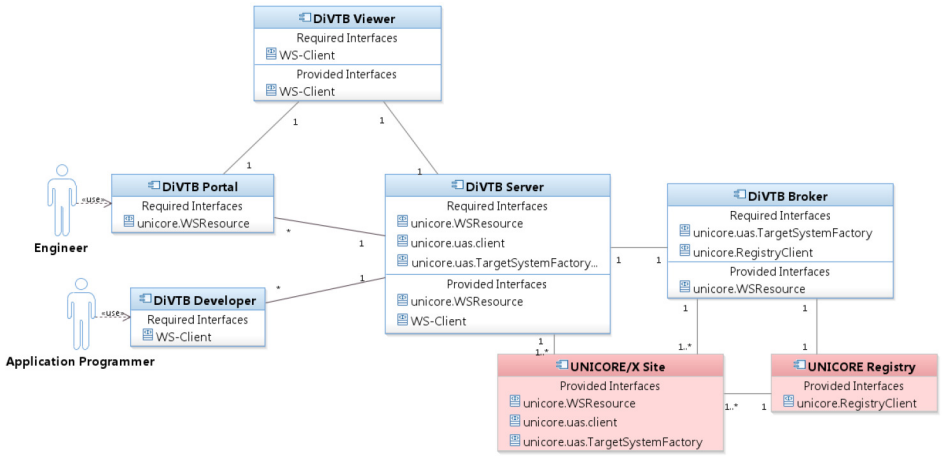

Figure 3. The architecture of the DiVTB system.

Architecture of the DiVTB Broker shown in the Fig. 4. The DiVTB Broker consists of the following components.

- The Master accepts requests from the DiVTB Server and creates an instance of a

WSResource that is called a Brokered Workflow.

- Each Brokered Workflow processes one request. It generates a list of required resources to perform a workflow and make the reservation of resources.
- The Resource Manager works with the Resource Database which contains the information about target systems and reserved resources.
- The Statistics Manager supports the Statistics Database which contains the information about tasks statistics (the execution time, the amount of transmitted data, the types of graphs).
- The Collector works independently from the DiVTB Broker and carries out the collection of the information for the Resource Database.

The DiVTB Broker creates a schedule for an initial graph using by the Resource Database and the Statistics Database.

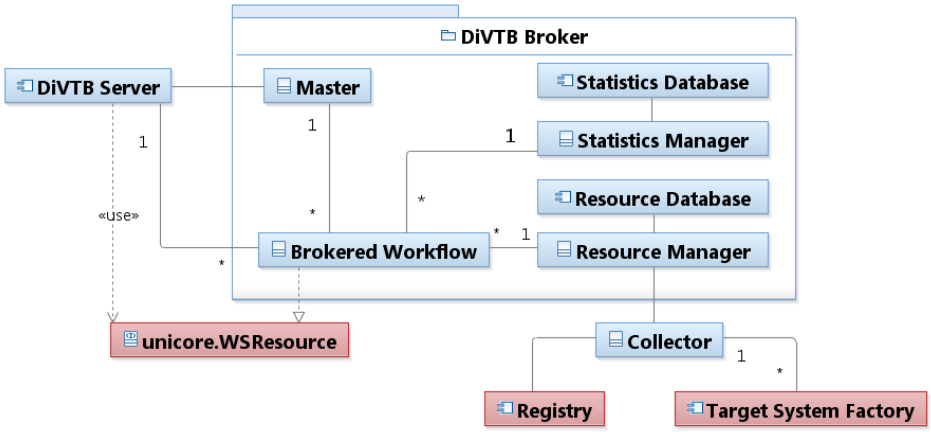

Figure 4. The architecture of the DiVTB Broker.

5 Conclusions

A developed mathematical model allows to describe a workflow as a job graph and to involve the use of vertex clustering; the ability to specify an execution time of each task and its upper limit of linear scalability; the ability to set the number of processor cores for each compute nodes. This mathematical model is a universal one, because it provides an opportunity to describe already existing and new algorithms for clustering, as well as new algorithms for scheduling of resources in distributed computing environments.

The POS (Problem-Oriented Scheduling) algorithm for distributed cluster computing environments allows us to schedule one task to run on multiple cores with the limitations

on its scalability. This scheduling algorithm is designed to create a system of intelligent building and scheduling a workflow in order to increase the efficiency of the supercomputer complexes with cluster architectures.

Acknowledgements

The present work was supported by Russian Foundation for Basic Research (RFBR), Research Project No. 14-07-31159-mol-a "My first grant".

References

1. S.J. Kim, *A general approach to multiprocessor scheduling*, Report TR-88-04. Department of Computer Science, University of Texas at Austin, 1988.
2. S.J. Kim, J.C. Browne, *A general approach to mapping of parallel computation upon multiprocessor architectures*, Proceedings of the International Conference on Parallel Processing, Volume 3, pp. 1-8, 1988.
3. V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*, The MIT Press, Cambridge, MA, P. 215, 1989.
4. A. Gerasoulis, T. Yang, *A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors*, J. Parallel and Distributed Computing **16(4)**, 276–291, 1992.
5. T. Yang, A. Gerasoulis, *DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors*, Proceedings of the IEEE Transactions on Parallel and Distributed Systems, Volume 5, No. 9. pp. 951-967, 1994.
6. A. Shamakina, *Brokering Service for Supporting Problem-Oriented Grid Environments*, Proceedings of the 2012 UNICORE Summit, IAS Series, Volume 15, pp. 67-75, 2012.
7. A.V. Shamakina, *Brokering Service for Supporting Problem-Oriented Grid Environments* J. Bulletin of South Ural State University. Series: Computational Mathematics and Informatics **46(305)**, 88–98, 2012 (in Russ.).
8. D.E. Knuth, *The Art of Computer Programming*, Volume 1: Fundamental Algorithms, 3rd Edition, P. 652, 1997.
9. V.V. Voevodin, *Mathematical models and methods in parallel processes*, Science, P. 296, 1986.
10. V.V. Voevodin, Vl.V. Voevodin. *Parallel computing*, St. Petersburg, BHV- Petersburg, P. 608, 2002.
11. G. Radchenko and E. Khudyakova, *Distributed Virtual Test Bed: an Approach to Integration of CAE Systems in UNICORE Grid Environment* Proceedings of the 36th International Convention MIPRO 2013, pp. 183-188, 2013.
12. G. Radchenko and E. Khudyakova, *A Service-Oriented Approach of Integration of Computer-Aided Engineering Systems in Distributed Computing Environments* Proceedings of the 2012 UNICORE Summit, IAS Series, Volume 15, pp. 57-66, 2012.
13. The UNICORE platform, see `www.unicore.eu/`
14. K. Benedyczak, P. Bała, S. van den Berghe, R. Menday, and B. Schuller, *Key aspects of the UNICORE 6 security model*, J. Future Generation Comp. Syst. **27(2)**, 195–201, 2011.

Schriften des Forschungszentrums Jülich
IAS Series

Band / Volume 14
Pedestrian fundamental diagrams:
Comparative analysis of experiments in different geometries
by J. Zhang (2012), xiii, 103 pages
ISBN: 978-3-89336-825-9
URN: urn:nbn:de:0001-2012102405

Band / Volume 15
UNICORE Summit 2012
Proceedings, 30 - 31 May 2012 | Dresden, Germany
edited by V. Huber, R. Müller-Pfefferkorn, M. Romberg (2012), iv, 143 pages
ISBN: 978-3-89336-829-7
URN: urn:nbn:de:0001-2012111202

Band / Volume 16
Design and Applications of an Interoperability Reference Model
for Production e-Science Infrastructures
by M. Riedel (2013), x, 270 pages
ISBN: 978-3-89336-861-7
URN: urn:nbn:de:0001-2013031903

Band / Volume 17
Route Choice Modelling and Runtime Optimisation
for Simulation of Building Evacuation
by A. U. Kemloh Wagoum (2013), xviii, 122 pages
ISBN: 978-3-89336-865-5
URN: urn:nbn:de:0001-2013032608

Band / Volume 18
Dynamik von Personenströmen in Sportstadien
by S. Burghardt (2013), xi, 115 pages
ISBN: 978-3-89336-879-2
URN: urn:nbn:de:0001-2013060504

Band / Volume 19
Multiscale Modelling Methods for Applications in Materials Science
by I. Kondov, G. Sutmann (2013), 326 pages
ISBN: 978-3-89336-899-0
URN: urn:nbn:de:0001-2013090204

Band / Volume 20
High-resolution Simulations of Strongly Coupled Coulomb Systems
with a Parallel Tree Code
by M. Winkel (2013), xvii, 196 pages
ISBN: 978-3-89336-901-0
URN: urn:nbn:de:0001-2013091802

Schriften des Forschungszentrums Jülich
IAS Series

Band / Volume 21
UNICORE Summit 2013
Proceedings, 18th June 2013 | Leipzig, Germany
edited by V. Huber, R. Müller-Pfefferkorn, M. Romberg (2013), iii, 94 pages
ISBN: 978-3-89336-910-2
URN: urn:nbn:de:0001-2013102109

Band / Volume 22
Three-dimensional Solute Transport Modeling in Coupled Soil and Plant Root Systems
by N. Schröder (2013), xii, 126 pages
ISBN: 978-3-89336-923-2
URN: urn:nbn:de:0001-2013112209

Band / Volume 23
Characterizing Load and Communication Imbalance in Parallel Applications
by D. Böhme (2014), xv, 111 pages
ISBN: 978-3-89336-940-9
URN: urn:nbn:de:0001-2014012708

Band / Volume 24
Automated Optimization Methods for Scientific Workflows in e-Science Infrastructures
by S. Holl (2014), xvi, 182 pages
ISBN: 978-3-89336-949-2
URN: urn:nbn:de:0001-2014022000

Band / Volume 25
Numerical simulation of gas-induced orbital decay of binary systems in young clusters
by A. C. Korntreff (2014), 98 pages
ISBN: 978-3-89336-979-9
URN: urn:nbn:de:0001-2014072202

Band / Volume 26
UNICORE Summit 2014
Proceedings, 24th June 2014 | Leipzig, Germany
edited by V. Huber, R. Müller-Pfefferkorn, M. Romberg (2014), iii, 60 pages
ISBN: 978-3-95806-004-3
URN: urn:nbn:de:0001-2014111408

Weitere ***Schriften des Verlags im Forschungszentrum Jülich*** unter
http://wwwzb1.fz-juelich.de/verlagextern1/index.asp

The UNICORE Grid technology provides a seamless, secure, and intuitive access to distributed Grid resources. UNICORE is a full-grown and well-tested Grid middleware system, which today is used in daily production worldwide. Beyond this production usage, the UNICORE technology serves as a solid basis in many European and International projects. In order to foster these ongoing developments, UNICORE is available as open source under BSD licence at http://www.unicore.eu.

The UNICORE Summit is a unique opportunity for Grid users, developers, administrators, researchers, and service providers to meet and share experiences, present past and future developments, and get new ideas for prosperous future work and collaborations. The UNICORE Summit 2014, the tenth in its series, has been held on June 24 2014 in Leipzig, Germany.

The proceedings at hand include a selection of 6 papers that show the spectrum of where and how UNICORE is used and further extended, especially with respect to data management and application support.

This publication was edited at the Jülich Supercomputing Centre (JSC) which is an integral part of the Institute for Advanced Simulation (IAS). The IAS combines the Jülich simulation sciences and the supercomputer facility in one organizational unit. It includes those parts of the scientific institutes at Forschungszentrum Jülich which use simulation on supercomputers as their main research methodology.

IAS Series
Volume 26
ISBN 978-3-95806-004-3

JÜLICH
FORSCHUNGSZENTRUM